Systems Programming in C++

Practical Course

Summer Term 2022

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. recent Ubuntu)
    - Ideally with root access
- Basic experience with Linux (in particular with shell)
- You are free to use your favorite OS, **we only support Linux**
    - Our CI server runs Linux
    - It will run automated tests on your submissions

Lecture & Tutorial

- Sessions
  - **Tuesday, 12:00 – 14:00, MI 02.11.018**
  - **Friday, 10:00 – 12:00, MI 02.11.018**
- Roughly 50% lectures and 50% tutorials
  - Lectures cover new content
  - Tutorials discuss assignments and any questions
  - Slides on https://db.in.tum.de/teaching/ss22/c++praktikum
- **Attendance is mandatory**
- Announcements on the website and through Mattermost

Preliminary Schedule

Day	Date	Session
Tue	26.04.2022	Lecture
Fri	29.04.2022	Lecture
Tue	03.05.2022	Lecture
Fri	06.05.2022	Lecture
Tue	10.05.2022	Tutorial
Fri	13.05.2022	Lecture
Tue	17.05.2022	Tutorial
Fri	20.05.2022	Lecture
Tue	24.05.2022	Tutorial
Fri	27.05.2022	Lecture
Tue	31.05.2022	Tutorial
Fri	03.06.2022	Lecture
Tue	07.06.2022	Holiday
Fri	10.06.2022	Tutorial

Day	Date	Session
Tue	14.06.2022	Lecture
Fri	17.06.2022	Tutorial
Tue	21.06.2022	Lecture
Fri	24.06.2022	Tutorial
Tue	28.06.2022	Lecture
Fri	01.07.2022	Tutorial
Tue	05.07.2022	Lecture
Fri	08.07.2022	Tutorial
Tue	12.07.2022	Lecture
Fri	15.07.2022	Lecture
Tue	19.07.2022	Tutorial
Fri	22.07.2022	Lecture
Tue	26.07.2022	Tutorial
Fri	29.07.2022	Combined

Assignments

- Brief non-coding quiz at the beginning of random lectures or tutorials
- Weekly programming assignments published after each lecture
  - No teams
  - Due approximately 9 days later (details published on each 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 21.08.2022 at 23:59 (three 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: https://db.in.tum.de/teaching/ss22/c++praktikum
- E-Mail: freitagm@in.tum.de, riegerm@in.tum.de, sichert@in.tum.de
- GitLab: https://gitlab.db.in.tum.de/cpplab22
- Mattermost: https://mattermost.db.in.tum.de/cpplab22

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, C++17)
- Current standard: C++20
- Next standard: C++23

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)

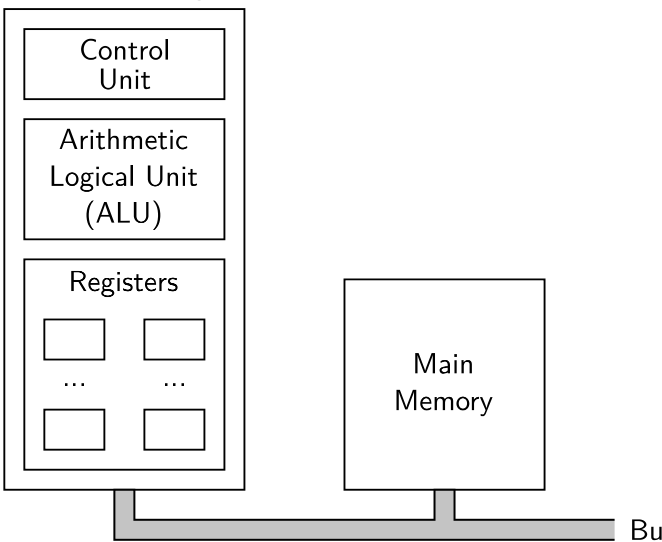
"Brains" of the computer

- 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)



Components of a CPU

Control Unit

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

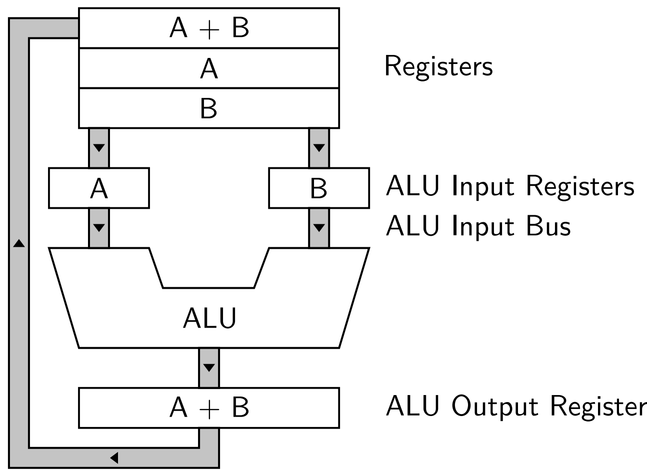
Arithmetic Logical Unit (ALU)

- 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 the 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

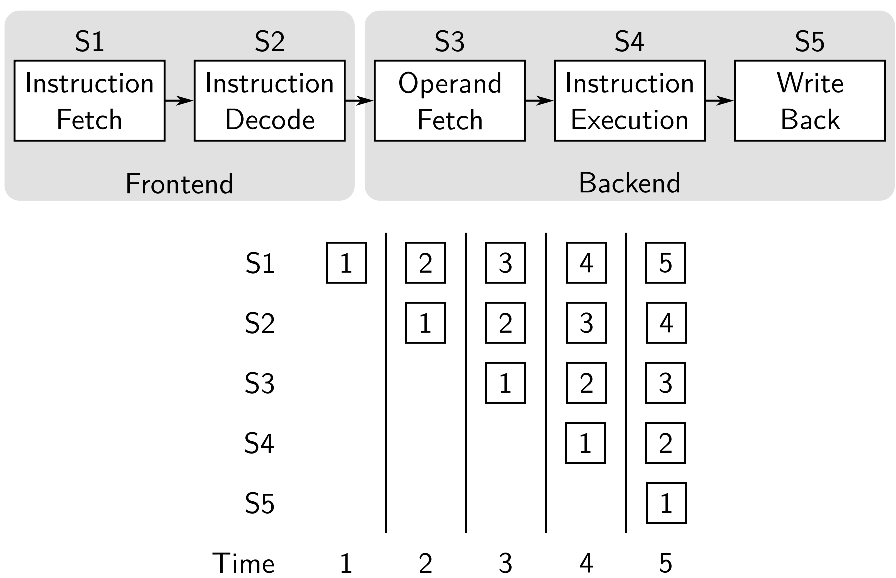
Instruction Prefetching

- 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 µops / 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

Idea: Execute more than one instruction per cycle

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

Superscalar architectures

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

Branch Prediction and Out-Of-Order Execution

The pipeline frontend requires branch prediction

- "Guess" which branches will be taken e.g. in `if`-statements
- Speculatively issue corresponding microoperations to pipeline backend
- Discard results if prediction did not come true
- Can heavily affect program performance

Microoperations may be executed out-of-order by the pipeline backend

- Effects of independent instructions may become visible in arbitrary order
- Order does not necessarily match instruction order in assembly
- Superscalar architectures require independent instructions for maximum performance

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
- CPU-local caches require highly sophisticated cache-coherency protocols

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

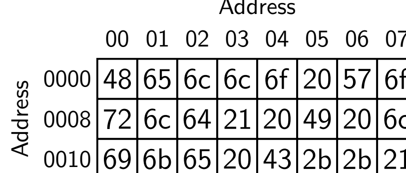
Addresses are memory words themselves

- 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
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
0000	H	e	l	l	o		W	o
0008	r	l	d	!		I		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 ⇒ **big-endian**
- Least significant byte first ⇒ **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: $\mathbf{42}_{10}$

16-bit integer at address 00: $\mathbf{0}_{10}$

00	01	02	03
00	2a	00	00

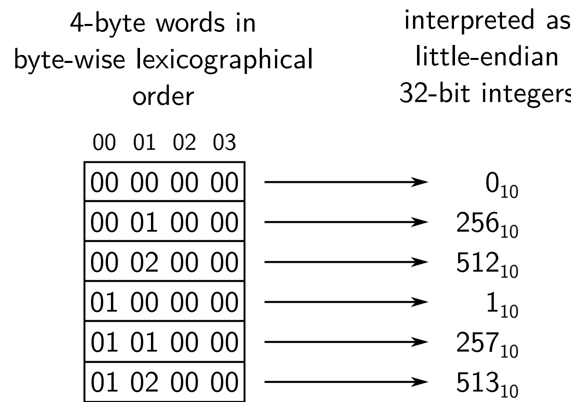
16-bit integer at address 00: $\mathbf{42}_{10}$

32-bit integer at address 00: $\mathbf{2{,}752{,}512}_{10}$

Byte Ordering (3)

Little-endian byte ordering can lead to unexpected results

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



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

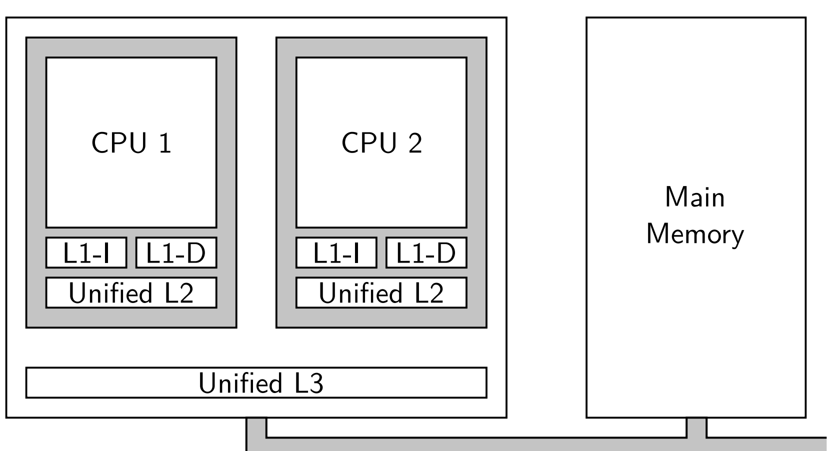
- CPU transparently keeps recently accessed data in cache (temporal locality)
- Memory is divided into blocks (**cache lines**)
- Whenever a memory cell is referenced, load the entire corresponding cache line into the cache (spatial locality)
- Requires specialized eviction strategy

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

Cache Memory on Multiprocessor Systems

Modern processors usually use a write-back strategy

- Writes to memory initially only change CPU-local caches (x86)
- Changes are propagated to main memory at some later time

Unpleasant side-effects on multiprocessor systems

- Memory reads and writes are ordered only within a single CPU
- Changes may become visible in arbitrary order on other CPUs
- Requires special programming models to maintain consistency

Assembly Language (1)

A basic understanding of assembly is immensely helpful when learning C++

- Understand how C++ features map to assembly
- Understand the close connection between C++ and low-level code
- (Sometimes) C++ design decisions become easier to understand
- (Sometimes) helps visualize what a piece of C++ code is doing

A basic understanding of assembly is immensely helpful when writing C++

- Ensure that you get the performance you expect from your code
- Ensure that you get the behavior you expect from your code
- Ensure that the compiler is doing what you expect it to do

Assembly Language (2)

Basic program structure

- Series of mnemonic processor instructions (e.g. `movl`, `addl`)
- Instructions usually operate on one or more operands
- Operands are usually registers, constants, or memory addresses

Example

```
movl    %edi, -4(%rbp)    # move data from register to memory
movl    -4(%rbp), %eax    # move data from memory to register
shll    $1, %eax          # shift register content 1 bit to left
addl    $42, %eax         # add 42 to register content
```

Registers (1)

Data is manipulated in the registers of a CPU

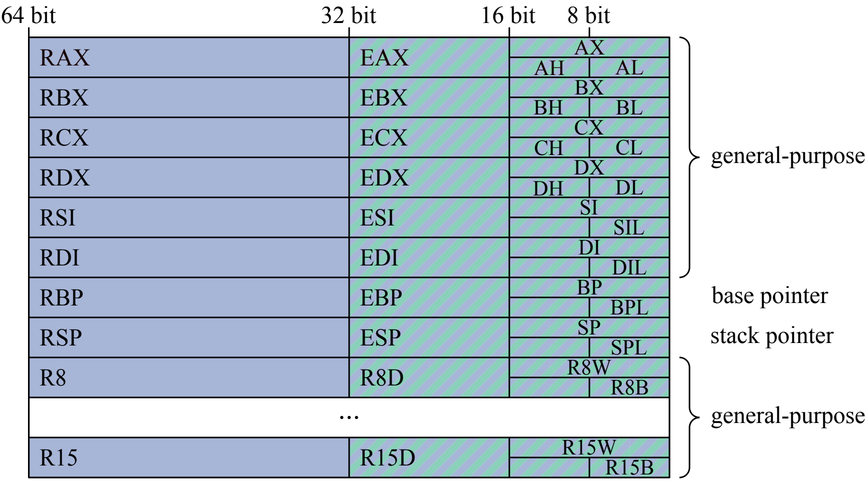
- CPUs contain a limited number of registers
- Registers are extremely fast in comparison to caches or main memory
- The compiler has to determine which variables to put into registers
- If not enough registers are available, variables are spilled into main memory

Assembly instructions usually manipulate data in registers

- Registers are referenced in assembly through their names (e.g. `eax`)
- Data transfer between memory and registers is explicit in assembly
- Some registers are used for specific purposes (e.g. `rip` for storing the instruction pointer)

Registers (2)

Important registers on x86-64



Godbolt Compiler Explorer

The Compiler Explorer created by Matt Godbolt is an invaluable tool

- Allows interactive viewing of the assembly generated by various C++ compilers
- We host an instance at https://compiler.db.in.tum.de/
- We encourage you to play with the tool throughout this course

Introduction to the C++ Ecosystem

Hello World in C++

myprogram.cpp

```cpp
#include <iostream>
int main(int argc, char** argv) {
    std::cout << "Hello " << argv[1] << '!' << std::endl;
    return 0;
}
```

```
$ c++ -std=c++20 -Wall -Werror -o myprogram ./myprogram.cpp
$ ./myprogram World
Hello World!
```

Generating an Executable Program

- Programs that transform C++ files into executables are called *compilers*
- Popular compilers: gcc (GNU), clang (llvm)
- Minimal example to compile the hello world program with gcc:

```
$ g++ -o myprogram ./myprogram.cpp
```

- Internally, the compiler is divided into:
  - Preprocessor
  - Compiler
  - Linker

Compiler Flags

General syntax to run a compiler: `c++ [flags] -o output inputs...`
Most common flags:

`-std=c++20`	Set C++ standard version
`-O0`	no optimization
`-O1`	optimize a bit, assembly mostly readable
`-O2`	optimize more, assembly not readable
`-O3`	optimize most, assembly not readable
`-Os`	optimize for size, similar to `-O3`
`-Wall`	Enable most warnings
`-Wextra`	Enable warnings not covered by `-Wall`
`-Werror`	Treat all warnings as errors
`-march=native`	Enable optimizations supported by your CPU
`-g`	Enable debug symbols

make

- C++ projects usually consist of many `.cpp` (*implementation files*) and `.hpp` (*header files*) files
- Each implementation file needs to be compiled into an object file first, then all object files must be linked
- Very repetitive to do this by hand
- When one `.cpp` file changes, only the corresponding object file should be recompiled, not all
- When one `.hpp` file changes, only implementation files that use it should be recompiled
- make is a program that can automate this
- Requires a `Makefile`
- GNU make manual: `https://www.gnu.org/software/make/manual/make.html`

Basic Makefile

- Makefiles consist of *rules* and contain *variables*
- Each rule has a *target*, *prerequisites*, and a *recipe*
- Recipes are only executed when the prerequisites are newer than the target or when the target does not exist
- Note: The indentation in Makefiles must be exactly one tab character, no spaces!

Makefile

```
CONTENT="test 123"  # set the variable CONTENT
# rule and recipe to generate the target file foo
foo:
	echo $(CONTENT) > foo
# $^ always contains all prerequisites ("foo baz" here)
# $< contains only the first prerequisite ("foo" here)
# $@ contains the target ("bar" here)
bar: foo baz
	cat $^ > $@
```

make and Timestamps

- make uses timestamps of files to decide when to execute recipes
- When any prerequisite file is newer than the target → execute recipe

```
$ make foo  # the file foo does not exist yet
echo "test 123"   > foo
$ make foo  # now foo exists
make: 'foo' is up to date.
$ make bar  # bar requires baz which doesn't exist
make: *** No rule to make target 'baz', needed by 'bar'.  Stop.
$ touch baz # create the file baz
$ make bar
cat foo baz > bar
$ make bar  # bar exists, nothing to do
make: 'bar' is up to date.
$ touch baz # update timestamp of file baz
$ make bar  # now the recipe for bar is executed again
cat foo baz > bar
```

Advanced Makefile

- Recipes are usually the same for most files
- *Pattern rules* can be used to reuse a recipe for multiple files

Makefile

```
CXX?=g++  # set CXX variable only if it's not set
CXXFLAGS+= -O3 -Wall -Wextra  # append to CXXFLAGS
SOURCES=foo.cpp bar.cpp
%.o: %.cpp  # pattern rule to make .o files out of .cpp files
	$(CXX) $(CXXFLAGS) -c -o $@ $<
# use a substitution reference to get .o file names
myprogram: myprogram.o $(SOURCES:.cpp=.o)
	$(CXX) $(CXXFLAGS) -o $@ $^
```

```
$ make  # executes the first (non-pattern) rule
g++ -O3 -Wall -Wextra   -c -o myprogram.o myprogram.cpp
g++ -O3 -Wall -Wextra   -c -o foo.o foo.cpp
g++ -O3 -Wall -Wextra   -c -o bar.o bar.cpp
g++ -O3 -Wall -Wextra   -o myprogram myprogram.o foo.o bar.o
```

CMake

- `make` prevents writing many repetitive compiler commands
- Still, extra flags must be specified manually (e.g. `-l` to link an external library)
- On different systems the same library may require different flags
- CMake is a tool specialized for C and C++ projects that uses a `CMakeLists.txt` to generate `Makefiles` or files for other build systems (e.g. ninja, Visual Studio)
- Also, the C++ IDE CLion uses CMake internally
- `CMakeLists.txt` consists of a series of *commands*
- CMake Reference Documentation: `https://cmake.org/cmake/help/latest/`

Basic CMakeLists.txt

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(myprogram)
set(MYPROGRAM_FILES sayhello.cpp saybye.cpp)
add_executable(myprogram myprogram.cpp ${MYPROGRAM_FILES})
```

```
$ mkdir build; cd build  # create a separate build directory
$ cmake ..  # generate Makefile from CMakeLists.txt
-- The C compiler identification is GNU 8.2.1
-- The CXX compiler identification is GNU 8.2.1
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/X/myproject/build
$ make
Scanning dependencies of target myprogram
[ 25%] Building CXX object CMakeFiles/myprogram.dir/myprogram.cpp.o
[ 50%] Building CXX object CMakeFiles/myprogram.dir/sayhello.cpp.o
[ 75%] Building CXX object CMakeFiles/myprogram.dir/saybye.cpp.o
[100%] Linking CXX executable myprogram
```

CMake Commands

`cmake_minimum_required(VERSION 3.10)`
: Require a specific cmake version.

`project(myproject)`
: Define a C/C++ project with the name “myproject”, required for every project.

`set(FOO a b c)`
: Set the variable `${FOO}` to be equal to `a b c`.

`add_executable(myprogram a.cpp b.cpp)`
: Define an executable to be built that consists of the source files `a.cpp` and `b.cpp`.

`add_library(mylib a.cpp b.cpp)`
: Similar to `add_executable()` but build a library.

`add_compile_options(-Wall -Wextra)`
: Add `-Wall -Wextra` to all invocations of the compiler.

`target_link_library(myprogram mylib)`
: Link the executable or library `myprogram` with the library `mylib`.

CMake Variables

CMake has many variables that influence how the executables and libraries are built. They can be set in the `CMakeLists.txt` with `set()`, on the command line with `cmake -D FOO=bar`, or with the program `ccmake`.

`CMAKE_CXX_STANDARD=20`
: Set the C++ to standard to C++20, effectively adds `-std=c++20` to the compiler flags.

`CMAKE_CXX_COMPILER=clang++`
: Set the C++ compiler to `clang++`.

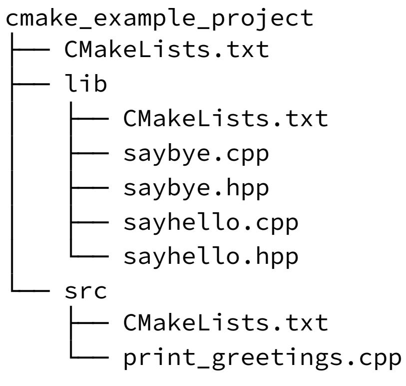
`CMAKE_BUILD_TYPE=Debug`
: Set the "build type" to `Debug`. Other possible values: `Release`, `RelWithDebInfo`. This mainly affects the optimization compiler flags.

`CMAKE_CXX_FLAGS(_DEBUG/_RELEASE)=-march=native`
: Add `-march=native` to all compiler invocations (or only for the `Debug` or `Release` build types).

Subdirectories with CMake

- Larger C++ projects are usually divided into subdirectories
- CMake allows the `CMakeLists.txt` to also be divided into the subdirectories
- A subdirectory can have its own `CMakeLists.txt` (without the `project()` command)
- The "main" `CMakeListst.txt` can then include the subdirectory with `add_subdirectory(subdir)`

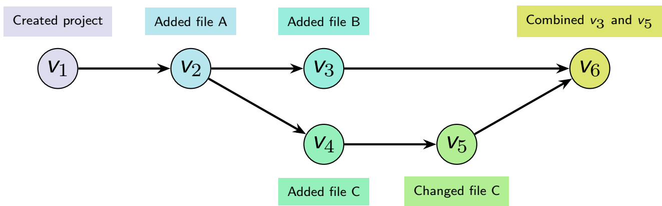
Complete CMake Example



- This project contains the library `greetings` and the executable `print_greetings`
- The library consists of the files `sayhello.cpp` and `saybye.cpp`
- You can find this project in our Gitlab

Version Control Systems (VCS)

- Code projects evolve gradually
- Incremental changes, also called *versions*, should be tracked to allow:
  - Documentation of the project history
  - Selective inspection/modification of specific versions
  - Efficient collaboration when working in a team
- Version Control Systems (VCS) manage versions, usually represent them in a directed acyclic graph



Git

- Many VCS exist, Git is a very popular one: Used by Linux, GCC, LLVM, etc.
- Git in particular has the following advantages compared to other version control systems (VCS):
  - Open source (LGPLv2.1)
  - Decentralized, i.e., no server required
  - Efficient management of *branches* and *tags*
- All Git commands are document with man-pages (e.g. type `man git-commit` to see documentation for the command `git commit`)
- Pro Git book: `https://git-scm.com/book`
- Git Reference Manual: `https://git-scm.com/docs`

Git Concepts

Repository: A collection of Git objects (*commits* and *trees*) and references (*branches* and *tags*).

Branch: A named reference to a *commit*. Every repository usually has at least the `master` branch and contains several more branches, like `fix-xyz` or `feature-abc`.

Tag: A named reference to a *commit*. In contrast to a branch a tag is usually set once and not changed. A branch regularly gets new commits.

Commit: A snapshot of a *tree*. Identified by a SHA1 hash. Each commit can have multiple parent commits. The commits form a directed acyclic graph.

Tree: A collection of files (not directories!) with their path and other metadata. This means that Git does *not* track empty directories.

Creating a Git Repository

`git init`
: Initialize a Git repository

`git config --global user.name <name>`
: Sets the name that will be used in commits

`git config --global user.email <email>`
: Sets the e-mail address that will be used in commits

`git status`
: Shows information about the repository

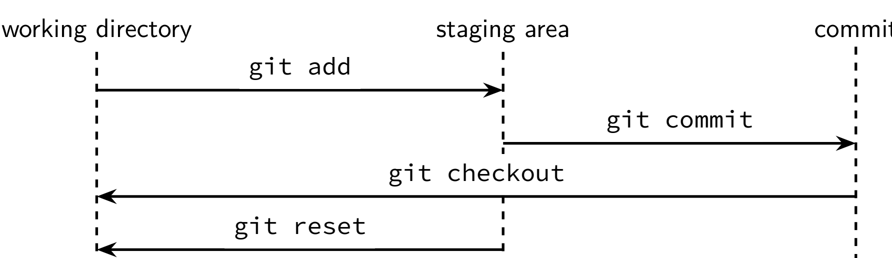
```
$ mkdir myrepo && cd myrepo
$ git init
Initialized empty Git repository in /home/X/myrepo/.git/
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Git Working Directory and Staging Area

When working with a Git repository, changes can live in any of the following places:

- In the working directory (when you edit a file)
- In the staging area (when you use `git add`)
- In a commit (after a `git commit`)

Once a change is in a commit and it is referenced by at least one branch or tag, you can always restore it even if you remove the file.



Committing Changes

`git add [-p] <path>...`
: Add changes to the staging area. Changes can be selected interactively when the -p option is used.

`git reset [-p] <path>...`
: Remove changes from the staging area without directly modifying the files. Can also be done interactively with -p.

`git commit`
: Take all changes from the staging area and turn them into a commit. Includes a commit message and author and date information. The parent of the new commit is set to the newest commit of the current branch. Then, the current branch is updated to point to the new commit.

`git checkout -- <path>...`
: Remove changes from the working directory by overwriting the given files or directories with their committed versions.

Inspecting the Commit History (1)

`git log [<branch>]`
View the commit history of the current (or another) branch.

`git show [<commit>]`
Show the changes introduced by the last (or the given) commit.

- "Browsing" the commit history with Git alone usually requires you to know the commands that list commits, show changes, etc., and execute several of them.
- There is a program called `tig` that provides a text-based interface where you can scroll through branches, commits, and changes.
- Running `tig` without arguments shows an overview of the current branch.
- `tig` also understands the subcommands `tig status`, `tig log`, and `tig show`, which take the same arguments as the `git` variants

Inspecting the Commit History (2)

`git diff`
: View the changes in the working directory (without the staging area).

`git diff --staged`
: View the changes in the staging area (without the working directory).

`git diff HEAD`
: View the changes in the working directory and the staging area.

`git diff branch1..branch2`
: View the changes between two branches (or tags, commits).

Example output of `git diff`

```
diff --git a/foo b/foo
index e965047..980a0d5 100644
--- a/foo
+++ b/foo
@@ -1 +1 @@
-Hello
+Hello World!
```

Working with Branches and Tags

`git branch`
: Show all branches and which one is active.

`git branch <name>`
: Create a new branch that points to the current commit (HEAD).

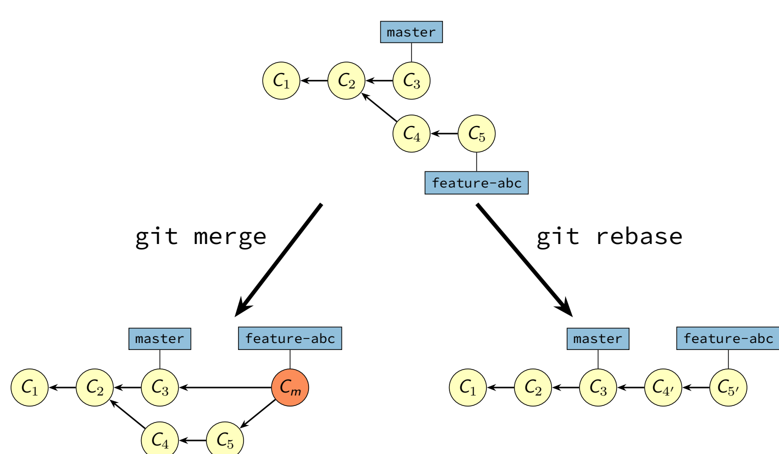
`git checkout <name>`
: Switch to another branch, i.e. change all files in the working directory so that they are equal to the tree of the other branch.

`git checkout -b <name>`
: Create a branch and switch to it.

`git tag`
: Show all tags.

`git tag [-s] <name>`
: Create a new tag that points to the current commit. Is signed with PGP when -s is given.

Modifying the Commit History (overview)



Modifying the Commit History

`git merge <branch>...`

- Combines the current branch and one or more other branches with a special *merge commit*
- The merge commit has the latest commit of all merged branches as parent
- No commit is modified

`git rebase <branch>`

- Start from the given branch and reapply all diverging commits from the current branch one by one
- All diverging commits are changed (they get a new parent) so their SHA1 hash changes as well

Dealing with Merge Conflicts

- Using `merge` or `rebase` may cause *merge conflicts*
- This happens when two commits are merged that contain changes to the same file
- When a merge conflict happens, Git usually tells you:

```
$ git merge branch2
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

- `git status` also shows additional information on how to proceed
- To fix the conflict you have to manually fix all conflicting files. Git inserts markers in the files to show where the conflicts arose:

foo

```
<<<<<<< HEAD
Hello World!
=======
Hello You!
>>>>>>> branch2
```

Undoing Committed Changes

: This operation may potentially irrevocably remove data

git revert <commit>
Create a new commit that is the "inverse" of the specified commit.

git reset <commit>
Reset the current branch to point to the given commit. No files are changed.

git reset --hard <commit> ⚠
Reset the current branch to point to the given commit. All files in the working directory are overwritten.

git rebase -i <commit>
Show all commits from the given one up to the current one and potentially remove individual commits.

git reflog
Shows a history of SHA1 commit hashes that were added or removed. Allows to restore removed commits if they were not garbage collected yet.

Working with Remote Git Repositories

`git clone <url>`
: Download the repository with all its commits, tags, and branches from the url.

`git push`
: Upload the current branch to a remote repository.

`git push --force-with-lease` ⚠
: Override the current branch on the remote repository. This is necessary when the local and remote branches have diverging histories, e.g., after using `git rebase` or `git reset --hard`.

`git fetch`
: Download new commits, tags, and branches from a remote repository into an existing repository.

`git pull`
: Run `git fetch` and then update (i.e. `git merge`) the current branch to match the branch on the remote repository.

Finding out Who Wrote the Code

- Sometimes, especially when reading a new code base, you want to know which commit changed a certain line
- Also, sometimes you want to know *who* wrote a certain line

`git blame <filename>`

- Shows the given file with commit annotations
- Each line starts with the commit hash, the name of the author, and the commit date

`tig blame <filename>`

- Like `git blame` but with a nicer interface
- Allows to "re-blame" at a given line, i.e. showing the selected line in the version just before it was last modified
- `tig` can also be used with other git commands: `tig log`, `tig diff`, etc.

Special Files in Git

`.gitignore`

- `git status`, `git diff`, etc. usually look at all files in all subdirectories of the repository
- If files or directories should always be excluded (e.g. build or cache directories), they can be added to the `.gitignore` file
- This file contains one entry per line, lines starting with # are skipped:

`foo.txt`	Ignores all files named `foo.txt`
`/foo.txt`	Ignores only the file `foo.txt` in the top-level directory
`foo/`	Ignores all directories named foo and their contents
`*f*`	Ignores all files and directories that contain the letter f

`.git`

- This directory contains all commits, branches, etc.
- E.g., `.git/refs/heads` contains one file per branch
- If you remove this directory, all data is lost!

Basic C++ Syntax

Overview

Common set of basic features shared by a wide range of programming languages

- Built-in types (integers, characters, floating point numbers, etc.)
- Variables ("names" for entities)
- Expressions and statements to manipulate values of variables
- Control-flow constructs (`if`, `for`, etc.)
- Functions, i.e. units of computation

Supplemented by additional functionality

- Programmer-defined types (`struct`, `class`, etc.)
- Library functions

The C++ Reference Documentation

C++ is in essence a simple language

- Limited number of basic features and rules
- **But:** There is a corner case to most features and an exception to most rules
- **But:** Some features and rules are rather obscure

These slides will necessarily be inaccurate or incomplete at times

- https://en.cppreference.com/w/cpp provides an excellent and complete reference documentation of C++
- Every C++ programmer should be able to read and understand the reference documentation
- Slides that directly relate to the reference documentation contain the symbol with a link to the relevant webpage in the slide header

Look at these links and familiarize yourself with the reference documentation!

Comments

C++ supports two types of comments

- "C-style" or "multi-line" comments: `/* comment */`
- "C++-style" or "single-line" comments: `// comment`

Example

```
/* This comment is unnecessarily
   split over two lines */
int a = 42;

// This comment is also split
// over two lines
int b = 123;
```

Fundamental Types

C++ defines a set of primitive types

- Void type
- Boolean type
- Integer types
- Character types
- Floating point types

All other types are composed of these fundamental types in some way

Void Type

The void type has no values

- Identified by the C++ keyword `void`
- No objects of type `void` are allowed
- Mainly used as a return type for functions that do not return any value
- Pointers to `void` are also permitted

```
void* pointer;          // OK: pointer to void
void object;            // ERROR: object of type void
void doSomething() {    // OK: void return type
    // do something important
}
```

Boolean Type

The boolean type can hold two values

- Identified by the C++ keyword `bool`
- Represents the truth values `true` and `false`
- Quite frequently obtained from implicit automatic type conversion

```
bool condition = true;
// ...
if (condition) {
    // ...
}
```

Integer Types (1)

The integer types represent integral values

- Identified by the C++ keyword `int`
- Some properties of integer types can be changed through modifiers
- `int` keyword may be omitted if at least one modifier is used

Signedness modifiers

- `signed` integers will have signed representation (i.e. they can represent negative numbers)
- Since C++20 signed integers must use two's complement representation
- `unsigned` integers will have unsigned representation (i.e. they can only represent non-negative numbers)

Size modifiers

- `short` integers will be optimized for space (at least 16 bits wide)
- `long` integers will be at least 32 bits wide
- `long long` integers will be at least 64 bits wide

Integer Types (2)

Modifiers and the `int` keyword can be specified in any order

```
// a, b, c and d all have the same type
unsigned long long int a;
unsigned long long     b;
long unsigned int long c;
long long unsigned     d;
```

By default integers are `signed`, thus the `signed` keyword can be omitted

```
// e and f have the same type
signed int e;
int        f;
```

By convention modifiers are ordered as follows

1. Signedness modifier
2. Size modifier
3. (`int`)

Integer Type Overview

Overview of the integer types as specified by the C++ standard

Canonical Type Specifier	Minimum Width	Minimum Range
`short` `unsigned short`	16 bit	-2^{15} to $2^{15}-1$ 0 to $2^{16}-1$
`int` `unsigned`	16 bit	-2^{15} to $2^{15}-1$ 0 to $2^{16}-1$
`long` `unsigned long`	32 bit	-2^{31} to $2^{31}-1$ 0 to $2^{32}-1$
`long long` `unsigned long long`	64 bit	-2^{63} to $2^{63}-1$ 0 to $2^{64}-1$

The exact width of integer types is **not** specified by the standard!

Fixed-Width Integer Types

Sometimes we need integer types with a guaranteed width

- Use fixed-width integer types defined in <cstdint> header
- int8_t, int16_t, int32_t and int64_t for signed integers of width 8, 16, 32 or 64 bit, respectively
- uint8_t, uint16_t, uint32_t and uint64_t for unsigned integers of width 8, 16, 32 or 64 bit, respectively

Only defined if the C++ implementation directly supports the type

```
#include <cstdint>

long     a;  // may be 32 or 64 bits wide
int32_t b;  // guaranteed to be 32 bits wide
int64_t c;  // guaranteed to be 64 bits wide
```

Integer Type Guidelines

Use basic (i.e. non-fixed-width) integer types by default

- They guarantee a minimum range that can be supported
- Most of the time we do not need to know an exact maximum value
- Usually (`unsigned`) `int` or `long` are a reasonable choice

Only use fixed-width integer types where absolutely required

- E.g. in data structures that need to have deterministic fixed size
- E.g. in library calls
- E.g. for bitwise operations that rely on masks, shifts etc.

Do not prematurely optimize for space consumption

- Registers on modern CPUs are likely to be 64 bit wide anyway
- Most of the time a program only becomes susceptible to overflow bugs if narrow integer types are used without good reason

Character Types

Character types represent character codes and (to some extent) integral values

- Identified by C++ keywords `signed char` and `unsigned char`
- Minimum width is 8 bit, large enough to represent UTF-8 eight-bit code units
- The C++ type `char` may either be equivalent to `signed char` or `unsigned char`, depending on the implementation
- Nevertheless `char` is always a distinct type
- `signed char` and `unsigned char` are sometimes used to represent small integral values

Larger UTF characters are supported as well

- `char16_t` for UTF-16 character representation
- `char32_t` for UTF-32 character representation

Floating Point Types

Floating point types of varying precision

- `float` usually represents IEEE-754 32 bit floating point numbers
- `double` usually represents IEEE-754 64 bit floating point numbers
- `long double` is a floating point type with extended precision (varying width depending on platform and OS, usually between 64 bit and 128 bit)

Floating point types may support special values

- Infinity
- Negative zero
- Not-a-number

Implicit Conversions (1)

Type conversions may happen automatically

- If we use an object of type `A` where an object of type `B` is expected
- Exact conversion rules are highly complex (full details in the reference documentation)

Some common examples

- If one assigns an integral type to `bool` the result is `false` if the integral value is `0` and `true` otherwise
- If one assigns `bool` to an integral type the result is `1` if the value is `true` and `0` otherwise
- If one assigns a floating point type to an integral type the value is truncated
- If one assigns an out-of-range value to an unsigned integral type of width w, the result is the original value modulo 2^w

Implicit Conversions (2)

Example

```
uint16_t i = 257;
uint8_t j = i;  // j is 1

if (j) {
    /* executed if j is not zero */
}
```

Undefined Behavior (1)

In some situations the behavior of a program is not well-defined

- E.g. overflow of an unsigned integer is well-defined (see previous slide)
- **But:** Signed integer overflow results in **undefined behavior**
- We will encounter undefined behavior every once in a while

Undefined behavior falls outside the specification of the C++ standard

- The compiler is allowed to do anything when it encounters undefined behavior
- Fall back to some sensible default behavior
- Do nothing
- Print 42
- Do anything else you can think of

A C++ program must never contain undefined behavior!

Undefined Behavior (2)

Example

foo.cpp

```
int foo(int i) {
    if ((i + 1) > i)
        return 42;

    return 123;
}
```

foo.o

```
foo(int):
    movl    $42, %eax
    retq
```

Undefined Behavior (3)

Undefined behavior differs from unspecified or implementation-defined behavior

- Unspecified or implementation-defined behavior is still valid C++
- However its effects may be different across compilers
- Only implementation-defined behavior is required to be documented

Undefined behavior gives compilers more freedom for optimization

- They can assume that programs contain no undefined behavior
- E.g. makes it possible for the compiler to omit some checks

Example

- Out-of-bounds array accesses are undefined behavior
- Therefore, the compiler does not need to generate range checks for each array access

Variables

Variables need to be defined before they can be used

- Simple declaration: Type specifier followed by comma-separated list of declarators (variable names) followed by semicolon
- Variable names in a simple declaration may optionally be followed by an initializer

```
void foo() {
    unsigned i = 0, j;
    unsigned meaningOfLife = 42;
}
```

Variable Initializers (1)

Initialization provides an initial value at the time of object construction

1. `variableName(<expression>)`
2. `variableName = <expression>`
3. `variableName{<expression>}`

Important differences

- Options 1 and 2 simply assign the value of the expression to the variable, possibly invoking implicit type conversions
- Option 3 results in a compile error if implicit type conversions potentially result in loss of information

A declaration may contain no initializer

- Non-local variables are default-initialized (to zero for built-in types)
- Local variables are usually **not** default-initialized

Accessing an uninitialized variable is **undefined behavior**

Variable Initializers (2)

```
double a = 3.1415926;
double b(42);
unsigned c = a;  // OK: c == 3
unsigned d(b);   // OK: d == 42
unsigned e{a};   // ERROR: potential information loss
unsigned f{b};   // ERROR: potential information loss
```

Initializers may be arbitrarily complex expressions

```
double pi = 3.1415926, z = 0.30, a = 0.5;
double volume(pi * z * z * a);
```

Integer Literals

Integer literals represent constant values embedded in the source code

- Decimal: `42`
- Octal: `052`
- Hexadecimal: `0x2a`
- Binary: `0b101010`

The following suffixes may be appended to a literal to specify its type

- `unsigned` suffix: `42u` or `42U`
- Long suffixes:
  - `long` suffix: `42l` or `42L`
  - `long long` suffix: `42ll` or `42LL`
- Both suffixes can be combined, e.g. `42ul`, `42ull`

Single quotes may be inserted between digits as a separator

- e.g. `1'000'000'000'000ull`
- e.g. `0b0010'1010`

Floating-point literals

Floating-point literals represent constant values embedded in the source code

- Without exponent: `3.1415926`, `.5`
- With exponent: `1e9`, `3.2e20`, `.5e-6`

One of the following suffixes may be appended to a literal to specify its type

- `float` suffix: `1.0f` or `1.0F`
- `long double` suffix: `1.0l` or `1.0L`

Single quotes may be inserted between digits as a separator

- e.g. `1'000.000'001`
- e.g. `.141'592e12`

Character Literals

Character literals represent constant values embedded in the source code

- Any character from the source character set except single quote, backslash and newline, e.g. `'a'`, `'b'`, `'€'`
- Escape sequences, e.g. `'\''`, `'\\'`, `'\n'`, `'\u1234'`

One of the following prefixes may be prepended to a literal to specify its type

- UTF-8 prefix: `u8'a'`, `u8'b'`
- UTF-16 prefix: `u'a'`, `u'b'`
- UTF-32 prefix: `U'a'`, `U'b'`

Const & Volatile Qualifiers (1)

Any type `T` in C++ (except function and reference types) can be *cv-qualified*

- const-qualified: `const T`
- volatile-qualified: `volatile T`
- cv-qualifiers can appear in any order, before or after the type

Semantics

- `const` objects cannot be modified
- Any read or write access to a `volatile` object is treated as a visible side effect for the purposes of optimization
- `volatile` should be avoided in most cases (it is likely to be deprecated in future versions of C++)
- Use *atomics* instead

Const & Volatile Qualifiers (2)

Only code that contributes to observable side-effects is emitted

```
int main() {
    int a = 1;  // will be optimized out
    int b = 2;  // will be optimized out
    volatile int c = 42;
    volatile int d = c + b;
}
```

Possible x86-64 assembly (compiled with `-O1`)

```
main:
    movl    $42, -4(%rsp)    # volatile int c = 42
    movl    -4(%rsp), %eax   # volatile int d = c + b;
    addl    $2, %eax         # volatile int d = c + b;
    movl    %eax, -8(%rsp)   # volatile int d = c + b;
    movl    $0, %eax         # implicit return 0;
    ret
```

Expression Fundamentals

C++ provides a rich set of operators

- Operators and operands can be composed into expressions
- Most operators can be overloaded for custom types

Fundamental expressions

- Variable names
- Literals

Operators act on a number of operands

- Unary operators: E.g. negation (−), address-of (&), dereference (⋆)
- Binary operators: E.g. equality (==), multiplication (⋆)
- Ternary operator: `a ? b : c`

Value Categories

Each expression in C++ is characterized by two independent properties

- Its *type* (e.g. `unsigned`, `float`)
- Its *value category*
- Operators may require operands of certain value categories
- Operators result in expressions of certain value categories

Broadly (and inaccurately) there are two value categories: *lvalues* and *rvalues*

- lvalues refer to the identity of an object
- rvalues refer to the value of an object
- Modifiable lvalues can appear on the left-hand side of an assignment
- lvalues and rvalues can appear on the right-hand side of an assignment

C++ actually has a much more sophisticated taxonomy of expressions

- Will (to some extent) become relevant later during the course

Arithmetic Operators (1)

Operator	Explanation
`+a`	Unary plus
`-a`	Unary minus
`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Modulo
`~a`	Bitwise NOT
`a & b`	Bitwise AND
`a | b`	Bitwise OR
`a ^ b`	Bitwise XOR
`a << b`	Bitwise left shift
`a >> b`	Bitwise right shift

C++ arithmetic operators have the usual semantics

Arithmetic Operators (2)

Incorrectly using the arithmetic operators can lead to undefined behavior, e.g.

- Signed overflow (see above)
- Division by zero
- Shift by a negative offset
- Shift by an offset larger than the width of the type

Logical and Relational Operators (1)

Operator	Explanation
`!a`	Logical NOT
`a && b`	Logical AND (short-circuiting)
`a || b`	Logical OR (short-circuiting)
`a == b`	Equal to
`a != b`	Not equal to
`a < b`	Less than
`a > b`	Greater than
`a <= b`	Less than or equal to
`a >= b`	Greater than or equal to
`a <=> b`	Three-way comparison

Most C++ logical and relational operators have the usual semantics

Logical and Relational Operators (2)

The three-way comparison (or spaceship) operator is a useful addition in C++20

- `(a <=> b) < 0` if `a < b`
- `(a <=> b) == 0` if `a == b`
- `(a <=> b) > 0` if `a > b`
- Can be generated by the compiler automatically in some cases
- Facilitates, for example, sorting values

Assignment Operators (1)

Operator	Explanation
`a = b`	Simple assignment
`a += b`	Addition assignment
`a -= b`	Subtraction assignment
`a *= b`	Multiplication assignment
`a /= b`	Division assignment
`a %= b`	Modulo assignment
`a &= b`	Bitwise AND assignment
`a |= b`	Bitwise OR assignment
`a ^= b`	Bitwise XOR assignment
`a <<= b`	Bitwise left shift assignment
`a >>= b`	Bitwise right shift assignment

Notes

- The left-hand side of an assignment operator must be a modifiable lvalue
- For built-in types `a OP= b` is equivalent to `a = a OP b` except that `a` is only evaluated once

Assignment Operators (2)

The assignment operators return a reference to the left-hand side

```
unsigned a, b, c;
a = b = c = 42;  // a, b, and c have value 42
```

Usually rarely used, with one exception

```
unsigned d;
if (d = computeValue()) {
    // executed if d is not zero
} else {
    // executed if d is zero
}

// unconditionally do something with d
```

Increment and Decrement Operators

Operator	Explanation
++a	Prefix increment
--a	Prefix decrement
a++	Postfix increment
a--	Postfix decrement

Return value differs between prefix and postfix variants

- Prefix variants increment or decrement the value of an object and return a *reference* to the result
- Postfix variants create a copy of an object, increment or decrement the value of the original object, and return the copy

Ternary Conditional Operator

Operator	Explanation
a ? b : c	Conditional operator

Semantics

- a is evaluated and converted to bool
- If the result was true, b is evaluated
- Otherwise c is evaluated

The type and value category of the resulting expression depend on the operands

```
int n = (1 > 2) ? 21 : 42;  // 1 > 2 is false, i.e. n == 42
int m = 42;
((n == m) ? m : n) = 21;    // n == m is true, i.e. m == 21

int k{(n == m) ? 5.0 : 21}; // ERROR: narrowing conversion
((n == m) ? 5 : n) = 21;    // ERROR: assigning to rvalue
```

Precedence and Associativity (1)

How to group multiple operators in one expression?

- Operators with higher precedence bind tighter than operators with lower precedence
- Operators with equal precedence are bound in the direction of their associativity
  - left-to-right
  - right-to-left
- Often grouping is not immediately obvious: **Use parentheses judiciously!**

Precedence and associativity do not specify evaluation order

- Evaluation order is mostly unspecified
- Generally, it is undefined behavior to refer to and change the same object within one expression

Precedence and Associativity (2)

In some situations grouping is obvious

```
int a = 1 + 2 * 3;  // 1 + (2 * 3), i.e. a == 7
```

However, things can get confusing really quickly

```
int b = 50 - 6 - 2;      // (50 - 6) - 2, i.e. b == 42
int c = b & 1 << 4 - 1; // b & (1 << (4 - 1)), i.e. c == 8

// real-world examples from libdcraw
diff = ((getbits(len-shl) << 1) + 1) << shl >> 1;      // ???
yuv[c] = (bitbuf >> c * 12 & 0xfff) - (c >> 1 << 11); // ???
```

Bugs like to hide in expressions without parentheses

```
// shift should be 4 if sizeof(long) == 4, 6 otherwise
unsigned shift = 2 + sizeof(long) == 4 ? 2 : 4; // buggy
```

Operator Precedence Table (1)

Prec.	Operator	Description	Associativity
1	::	Scope resolution	left-to-right
	a++ a--	Postfix increment/decrement	
	<type>()	Functional Cast	
2	<type>{}		left-to-right
	a()	Function Call	
	a[]	Subscript	
	. ->	Member Access	
	++a --a	Prefix increment/decrement	
	+a -a	Unary plus/minus	
	! ~	Logical/Bitwise NOT	
	(<type>)	C-style cast	
3	*a	Dereference	right-to-left
	&a	Address-of	
	sizeof	Size-of	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	

Operator Precedence Table (2)

Prec.	Operator	Description	Associativity
4	`.*` `->*`	Pointer-to-member	left-to-right
5	`a*b` `a/b` `a%b`	Multiplication/Division/Remainder	left-to-right
6	`a+b` `a-b`	Addition/Subtraction	left-to-right
7	`<<` `>>`	Bitwise shift	left-to-right
8	`<=>`	Three-way comparison	left-to-right
9	`<` `<=` `>` `>=`	Relational $<$ and $\leq$ Relational $>$ and $\geq$	left-to-right
10	`==` `!=`	Relational $=$ and $\neq$	left-to-right

Operator Precedence Table (3)

Prec.	Operator	Description	Associativity
11	`&`	Bitwise AND	left-to-right
12	`^`	Bitwise XOR	left-to-right
13	`|`	Bitwise OR	left-to-right
14	`&&`	Logical AND	left-to-right
15	`||`	Logical OR	left-to-right

16	`a?b:c`
	`throw`
	`=`
	`+=` `-=`
	`*=` `/=` `%=`
	`<<=` `>>=`
	`&=` `^=` `|=`
		Ternary conditional
		throw operator
		Direct assignment
		Compound assignment
		Compound assignment
		Compound assignment
		Compound assignment
			right-to-left

17	`,`	Comma	left-to-right

Simple Statements

Declaration statement: Declaration followed by a semicolon

```
int i = 0;
```

Expression statement: Any expression followed by a semicolon

```
i + 5;  // valid, but rather useless expression statement
foo();  // valid and possibly useful expression statement
```

Compound statement (blocks): Brace-enclosed sequence of statements

```
{                // start of block
    int i = 0;   // declaration statement
}                // end of block, i goes out of scope
int i = 1;       // declaration statement
```

Scope

Names in a C++ program are valid only within their *scope*

- The scope of a name begins at its point of declaration
- The scope of a name ends at the end of the relevant block
- Scopes may be shadowed resulting in discontiguous scopes (bad practice)

```
int a = 21;
int b = 0;
{
    int a = 1;      // scope of the first a is interrupted
    int c = 2;
    b = a + c + 39; // a refers to the second a, b == 42
}                   // scope of the second a and c ends
b = a;              // a refers to the first a, b == 21
b += c;             // ERROR: c is not in scope
```

If Statement (1)

Conditionally executes another statement

```
if (init-statement; condition)
    then-statement
else
    else-statement
```

Explanation

- If *condition* evaluates to `true` after conversion to `bool`, *then-statement* is executed, otherwise *else-statement* is executed
- Both *init-statement* and the else branch can be omitted
- If present, *init-statement* must be an expression or declaration statement
- *condition* must be an expression statement or a single declaration
- *then-statement* and *else-statement* can be arbitrary (compound) statements

If Statement (2)

The *init-statement* form is useful for local variables only needed inside the `if`

```
if (unsigned value = computeValue(); value < 42) {
    // do something
} else {
    // do something else
}
```

Equivalent formulation

```
{
    unsigned value = computeValue();
    if (value < 42) {
        // do something
    } else {
        // do something else
    }
}
```

If Statement (3)

In nested `if`-statements, the `else` is associated with the closest `if` that does not have an `else`

```
// INTENTIONALLY BUGGY!
if (condition0)
    if (condition1)
        // do something if (condition0 && condition1) == true
else
    // do something if condition0 == false
```

When in doubt, use curly braces to make scopes explicit

```
// Working as intended
if (condition0) {
    if (condition1)
    // do something if (condition0 && condition1) == true
} else {
    // do something if condition0 == false
}
```

Switch Statement (1)

Conditionally transfer control to one of several statements

```
switch (init-statement; condition)
    statement
```

Explanation

- *condition* may be an expression or single declaration that is convertible to an enumeration or integral type
- The body of a `switch` statement may contain an arbitrary number of `case` *`constant`*`:` labels and up to one `default:` label
- The constant values for all `case:` labels must be unique
- If *condition* evaluates to a value for which a `case:` label is present, control is passed to the labelled statement
- Otherwise, control is passed to the statement labelled with `default:`
- The `break;` statement can be used to exit the `switch`

Switch Statement (2)

Regular example

```
switch (computeValue()) {
    case 21:
        // do something if computeValue() was 21
        break;
    case 42:
        // do something if computeValue() was 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```

Switch Statement (3)

The body is executed sequentially until a `break;` statement is encountered

```
switch (computeValue()) {
    case 21:
    case 42:
        // do something if computeValue() was 21 or 42
        break;
    default:
        // do something if computeValue() was != 21 and != 42
        break;
}
```

Compilers may generate warnings when encountering such fall-through behavior

- Use special `[[fallthrough]];` statement to mark intentional fall-through

While Loop

Repeatedly executes a statement

```
while (condition)
    statement
```

Explanation

- Executes *statement* repeatedly until the value of *condition* becomes `false`. The test takes place before each iteration.
- *condition* may be an expression that can be converted to `bool` or a single declaration
- *statement* may be an arbitrary statement
- The `break;` statement may be used to exit the loop
- The `continue;` statement may be used to skip the remainder of the body

Do-While Loop

Repeatedly executes a statement

```
do
    statement
while (condition);
```

Explanation

- Executes *statement* repeatedly until the value of *condition* becomes `false`. The test takes place after each iteration.
- *condition* may be an expression that can be converted to `bool` or a single declaration
- *statement* may be an arbitrary statement
- The `break;` statement may be used to exit the loop
- The `continue;` statement may be used to skip the remainder of the body

While vs. Do-While

The body of a do-while loop is executed at least once

```
unsigned i = 42;

do {
    // executed once
} while (i < 42);

while (i < 42) {
    // never executed
}
```

For Loop (1)

Repeatedly executes a statement

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

Explanation

- Executes *init-statement* once, then executes *statement* and *iteration-expression* repeatedly until *condition* becomes `false`
- *init-statement* may either be an expression or declaration
- *condition* may either be an expression that can be converted to `bool` or a single declaration
- *iteration-expression* may be an arbitrary expression
- All three of the above statements may be omitted
- The `break;` statement may be used to exit the loop
- The `continue;` statement may be used to skip the remainder of the body

For Loop (2)

```
for (unsigned i = 0; i < 10; ++i) {
    // do something
}

for (unsigned i = 0, limit = 10; i != limit; ++i) {
    // do something
}
```

Beware of integral overflows (signed overflows are undefined behavior!)

```
for (uint8_t i = 0; i < 256; ++i) {
    // infinite loop
}

for (unsigned i = 42; i >= 0; --i) {
    // infinite loop
}
```

Basic Functions (1)

Functions in C++

- Associate a sequence of statements (the *function body*) with a name
- Functions may have zero or more *function parameters*
- Functions can be invoked using a function-call expression which initializes the parameters from the provided arguments

Informal function definition syntax

```
return-type name ( parameter-list ) {
    statement
}
```

Informal function call syntax

```
name ( argument-list );
```

Basic Functions (2)

Function may have void return type

```
void procedure(unsigned parameter0, double parameter1) {
    // do something with parameter0 and parameter1
}
```

Functions with non-void return type must contain a return statement

```
unsigned meaningOfLife() {
    // extremely complex computation
    return 42;
}
```

The return statement may be omitted in the main-function of a program (in which case zero is implicitly returned)

```
int main() {
    // run the program
}
```

Basic Functions (3)

Function parameters may be unnamed, in which case they cannot be used

```
unsigned meaningOfLife(unsigned /*unused*/) {
    return 42;
}
```

An argument must still be supplied when invoking the function

```
unsigned v = meaningOfLife();     // ERROR: expected argument
unsigned w = meaningOfLife(123);  // OK
```

Argument Passing

Argument to a function are passed **by value** in C++

```
unsigned square(unsigned v) {
    v = v * v;
    return v;
}

int main() {
    unsigned v = 8;
    unsigned w = square(v);  // w == 64, v == 8
}
```

C++ differs from other programming languages (e.g. Java) in this respect

- Parameters can *explicitly* be passed by reference
- Essential to keep argument-passing semantics in mind, especially when used-defined classes are involved

Default Arguments

A function definition can include default values for some of its parameters

- Indicated by including an initializer for the parameter
- After a parameter with a default value, all subsequent parameters must have default values as well
- Parameters with default values may be omitted when invoking the function

```
int foo(int a, int b = 2, int c = 3) {
    return a + b + c;
}

int main() {
    int x = foo(1);         // x == 6
    int y = foo(1, 1);      // y == 5
    int z = foo(1, 1, 1);   // z == 3
}
```

Function Overloading (1)

Several functions may have the same name (*overloaded*)

- Overloaded functions must have distinguishable parameter lists
- Calls to overloaded functions are subject to *overload resolution*
- Overload resolution selects which overloaded function is called based on a set of complex rules

Informally, parameter lists are distinguishable

- If they have a different number of non-defaulted parameters
- If they have at least one parameter with different type

Function Overloading (2)

Indistinguishable parameter lists (invalid C++)

```
void foo(unsigned i);
void foo(unsigned j);  // parameter names do not matter
void foo(unsigned i, unsigned j = 1);
void foo(uint32_t i);  // on x86_64
```

Valid example

```
void foo(unsigned i) { /* do something */ }
void foo(float f) { /* do something */ }

int main() {
    foo(1u);   // calls foo(unsigned)
    foo(1.0f); // calls foo(float)
}
```

Basic IO (1)

Facilities for printing to and reading from the console

- Use *stream objects* defined in `<iostream>` header
- `std::cout` is used for printing to console
- `std::cin` is used for reading from console

The left-shift operator can be used to write to `std::cout`

```
#include <iostream>
// ----------------------------------
int main() {
    unsigned i = 42;
    std::cout << "The value of i is " << i << std::endl;
}
```

Basic IO (2)

The right-shift operator can be used to read from `std::cin`

```
#include <iostream>
// ----------------------------------
int main() {
    std::cout << "Please enter a value: " << std::flush;
    unsigned v;
    std::cin >> v;
    std::cout << "You entered " << v << std::endl;
}
```

The `<iostream>` header is part of the C++ standard library

- Many more interesting and useful features
- More details later
- In the meantime: Read the documentation!

Code Formatting (1)

Projects should always use a uniform code style

- Consistent conventions for naming, documentation, etc.
- Some aspects of a uniform code style have to be implemented manually (e.g. naming conventions)

Automated code formatting can for example be performed with `clang-format`

- Widely available through package manager
- Highly configurable code formatting tool
- Configuration possible through `.clang-format` file
- Integrated in CLion

Code Formatting (2)

Basic `clang-format` usage

```
> clang-format -i <path-to-file>
```

Reformats a source file in-place

- Reads formatting rules from `.clang-format` file in the current directory
- Should usually reside in the source root for project-wide formatting rules
- CLion detects `.clang-format` files and uses them for formatting
- Can be verified by looking for "ClangFormat" in the status bar of CLion

Code Formatting (3)

We will provide you with a `.clang-format` file for now

- Contains (in our opinion) sensible formatting rules
- Please make sure that your submissions are formatted according to these rules
- But our formatting rules should not be seen as the single source of truth

Some high-level formatting guidelines should be universally followed

- Descriptive names for variables and functions
- Comments for complicated sections of code
- ...

Compiling C++ files

Hello World 2.0

In C++ the code is usually separated into *header files* (.h/.hpp) and *implementation files* (.cpp/.cc):

sayhello.hpp

```
#include <string_view>
void sayhello(std::string_view name);
```

sayhello.cpp

```
#include "sayhello.hpp"
#include <iostream>
void sayhello(std::string_view name) {
    std::cout << "Hello " << name << '!' << std::endl;
}
```

Other code that wants to use this function only has to include sayhello.hpp.

Compiler

Reminder: Internally, the compiler is divided into Preprocessor, Compiler, and Linker.

Preprocessor:

- Takes an input file of (almost) any programming language
- Handles all *preprocessor directives* (i.e., all lines starting with #) and *macros*
- Outputs the file without any preprocessor directives or macros

Compiler:

- Takes a preprocessed C++ (or C) file, called *translation unit*
- Generates and optimizes the machine code
- Outputs an *object file*

Linker:

- Takes multiple object files
- Can also take references to other libraries
- Finds the address of all symbols (e.g., functions, global variables)
- Outputs an executable file or a shared library

Preprocessor (1)

Preprocessor directive `#include`: Copies (**!**) the contents of a file into the current file.

Syntax:

- `#include "path"` where *path* is a relative path to the header file
- `#include <path>` like the first version but only *system directories* are searched for the *path*

In C++ usually only header files are included, never `.cpp` files!

Preprocessor (2)

Preprocessor directive `#define`: Defines a macro.

Syntax:

- `#define FOO`: Defines the macro `FOO` with no content
- `#define BAR 1`: Defines the macro `BAR` as 1

Before the compiler sees the file, all occurrences of `FOO` will be removed, `BAR` will be replaced with 1.

Note: Don't use this as "constant variables", use `constexpr` global variables instead!

Preprocessor (3)

Preprocessor directives `#ifdef`/`#ifndef`/`#else`/`#endif`: Removes all code up to the next `#else`/`#endif` if a macro is set (`#ifdef`) or not set (`#ifndef`)

Example:

```
#ifdef FOO
...
#endif
```

Mainly used for *header guards*.

Compiler

- Every *translation unit* (usually a `.cpp` file) results in exactly one object file (usually `.o`)
- References to external symbols (e.g., functions that are defined in another `.cpp`) are *not* resolved

mul.cpp

```
int add(int a, int b);
int mul(int a, int b) {
    if (a > 0) { return add(a, mul(a - 1, b)); }
    else { return 0; }
}
```

Assembly generated by the compiler:

```
_Z3mulii:
    testl   %edi, %edi
    jle     .L2
    pushq   %rbx
    movl    %edi, %ebx
    leal    -1(%rdi), %edi
    call    _Z3mulii
    movl    %ebx, %edi
    popq    %rbx
    movl    %eax, %esi
    jmp     _Z3addii@PLT
.L2:
    xorl    %eax, %eax
    ret
```

You can try this out yourself at `https://compiler.db.in.tum.de`

Linker

- The linker usually does not have to know about any programming language
- Still, some problems with your C++ code will only be found by the linker and not by the compiler (e.g., ODR violations)
- Most common error are missing symbols, happens either because you forgot to define a function or global variable, or forgot to add a library
- Popular linkers are: GNU `ld`, GNU `gold`, `lld` (by the LLVM project)

Compiler Flags (2)

- Preprocessor and linker are usually executed by the compiler
- There are additional compiler flags that can influence the preprocessor or the linker

Advanced flags:

`-E`	Run only preprocessor (outputs C++ file without preprocessor directives)
`-c`	Run only preprocessor and compiler (outputs object file)
`-S`	Run only preprocessor and compiler (outputs assembly as text)
`-g`	Add *debug symbols* to the generated binary
`-DFOO`	Defines the macro `FOO`
`-DFOO=42`	Defines the macro `FOO` with value 42
`-l<lib>`	Link library `<lib>` into executable
`-I<path>`	Also search `<path>` for `#included` files
`-L<path>`	Also search `<path>` for libraries specified with `-l`

Debugging C++ Programs with gdb

- Debugging by printing text is easy but most of the time not very useful
- Especially for multi-threaded programs a real debugger is essential
- For C++ the most used debugger is gdb ("GNU debugger")
- It is free and open-source (GPLv2)
- For the best debugging experience a program should be compiled without optimizations (`-O0`) and with debug symbols (`-g`)
- The debug symbols help the debugger to map assembly instructions to the source code that generated them
- The documentation for gdb can be found here: `https://sourceware.org/gdb/current/onlinedocs/gdb/`

gdb commands (1)

To start debugging run the command `gdb myprogram`. This starts a command-line interface wich expects one of the following commands:

`help`	Show general help or help about a command.
`run`	Start the debugged program.
`break`	Set a breakpoint. When the breakpoint is reached, the debugger stops the program and accepts new commands.
`delete`	Remove a breakpoint.
`continue`	Continue running the program after it stopped at a breakpoint or by pressing Ctrl + C .
`next`	Continue running the program until the next source line of the current function.
`step`	Continue running the program until the source line changes.
`nexti`	Continue running the program until the next instruction of the current function.
`stepi`	Execute the next instrution.
`print`	Print the value of a variable, expression or CPU register.

gdb commands (2)

`frame`	Show the currently selected *stack frame*, i.e. the current stack with its local variables. Usually includes the function name and the current source line. Can also be used to switch to another frame.
`backtrace`	Show all stack frames.
`up`	Select the frame from the next higher function.
`down`	Select the frame from the next lower function.
`watch`	Set a watchpoint. When the memory address that is watched is read or written, the debugger stops.
`thread`	Show the currently selected thread in a multi-threaded program. Can also be used to switch to another thread.

Most commands also have a short version, e.g., `r` for `run`, `c` for `continue`, etc.

Runtime Checks for Debugging

- Stepping though a buggy part of the program is often enough to identify the bug
- At least, it can help to narrow down the location of a bug
- Sometimes it is better to write code that checks if an invariant holds

The `assert` macro can be used for that:

- Defined in the `<cassert>` header
- Can be used to check a boolean expression
- Only enabled when the `NDEBUG` macro is *not* defined
- Automatically enabled in debug builds when using CMake

div.cpp

```
#include <cassert>
double div(double a, int b) {
    assert(b != 0);
    return a / b;
}
```

When this function is called with `b==0`, the program will crash with a useful error message.

Automatic Runtime Checks (“Sanitizers”)

- Modern compilers can automatically add several runtime checks, they are usually called *sanitizers*
- Most important ones:
  - Address Sanitizer (ASAN): Instruments memory access instructions to check for common bugs
  - Undefined-Behavior Sanitizer (UBSAN): Adds runtime checks to guard against many kinds of undefined behavior
- Because sanitizers add overhead, they are not enabled by default
- Should normally be used in conjunction with `-g` for debug builds
- Compiler option for gcc/clang: `-fsanitize=`<*sanitizer*>
  - `-fsanitize=address` for ASAN
  - `-fsanitize=undefined` for UBSAN
- Should be enabled by default in your debug builds, unless there is a very compelling reason against it

UBSAN Example

foo.cpp

```
#include <iostream>
int main() {
    int a; int b;
    std::cin >> a >> b;
    int c = a * b;
    std::cout << c << std::endl;
    return 0;
}
```

```
$ g++ -std=c++20 -g -fsanitize=undefined foo.cpp -o foo
$ ./foo
123456
789123
foo.cpp:7:9: runtime error: signed integer overflow: 123456 *
↪  789123 cannot be represented in type 'int'
-1362278720
```

Declarations and Definitions

Objects

One of the core concepts of C++ are objects.

- The main purpose of C++ programs is to interact with objects in order to achieve some goal
- Examples of objects are local and global variables
- Examples of concepts that are *not* objects are functions, references, and values

An object in C++ is a *region of storage* with certain properties:

- Size
- Alignment
- Storage duration
- Lifetime
- Type
- Value
- Optionally, a name

Storage Duration (1)

Every object has one of the following *storage durations*:

automatic:

- Objects with automatic storage duration are allocated at the beginning of the enclosing scope and deallocated automatically (i.e., it is not necessary to write code for this) at its end
- Local variables have automatic storage duration

static:

- Objects with static storage duration are allocated when the program begins (usually even before `main()` is executed!)
- They are deallocated automatically when the program ends
- All global variables have static storage duration
- The order of construction of different variables is not guaranteed → can easily lead to unexpected bugs. (See also: Static Initialization Order Fiasco).

Storage Duration (2)

thread:

- Objects with thread storage duration are allocated when a thread starts and deallocated automatically when it ends
- In contrast to objects with static storage duration, each thread gets its own copy of objects with thread storage duration

dynamic:

- Objects with dynamic storage duration are allocated and deallocated by using dynamic memory management
- Note: Deallocation must be done manually!

```
int foo = 1; // static storage duration
static int bar = 2; // static storage duration
thread_local int baz = 3; // thread storage duration
void f() {
    int x = 4; // automatic storage duration
    static int y = 5; // static storage duration
}
```

Lifetime

In addition to their storage duration objects also have a *lifetime* which is closely related. References also have a lifetime.

- The lifetime of an object or reference starts when it was fully *initialized*
- The lifetime of an object ends when its destructor is called (for objects of class types) or when its storage is deallocated or reused (for all other types)
- The lifetime of an object never exceeds its storage duration.
- The lifetime of a reference ends as if it were a “scalar” object (e.g. an `int` variable)

Generally, using an object outside of its lifetime leads to undefined behavior.

Lifetime issues are the main source of memory bugs!

- A C++ compiler can only warn about very basic lifetime errors
- If the compiler warns, always fix your code so that the warning disappears

Namespaces (1)

Larger projects may contain many names (functions, classes, etc.)

- Should be organized into logical units
- May incur name clashes
- C++ provides *namespaces* for this purpose

Namespace definitions

```
namespace identifier {
    namespace-body
}
```

Explanation

- *identifier* may be a previously unused identifier, or the name of a namespace
- *namespace-body* may be a sequence of declarations
- A name declared inside a namespace must be qualified when accessed from outside the namespace (`::` operator)

Namespaces (2)

Qualified name lookup

```
namespace A {
void foo() { /* do something */ }
void bar() {
    foo();  // refers to A::foo
}
}
namespace B {
void foo() { /* do something */ }
}
int main() {
    A::foo();  // qualified name lookup
    B::foo();  // qualified name lookup

    foo();     // ERROR: foo was not declared in this scope
}
```

Namespaces (3)

Namespaces may be nested

```
namespace A { namespace B {
void foo() { /* do something */ }
}}

// equivalent definition
namespace A::B {
void bar() {
    foo();  // refers to A::B::foo
}
}

int main() {
    A::B::bar();
}
```

Namespaces (4)

Code can become rather confusing due to large number of braces

- Use visual separators (comments) at sensible points
- (Optionally) add comments to closing namespace braces

```
//----------------------------------
namespace A::B {
//----------------------------------
void foo() {
    // do something
}
//----------------------------------
void bar() {
    // do something else
}
//----------------------------------
}  // namespace A::B
//----------------------------------
```

Namespaces (5)

- Always using fully qualified names makes code easier to read
- Sometimes it is obvious from which namespace the names come from in which case one prefers to use unqalified names
- For this `using` and `using namespace` can be used
- `using namespace X` imports *all* names from namespace X
- `using X::a` only imports the name a from X into the current namespace
- Should not be used in header files to not influence other implementation files

```
namespace A { int x; }
namespace B { int y; int z; }
using namespace A;
using B::y;
int main() {
    x = 1; // Refers to A::x
    y = 2; // Refers to B::y
    z = 3; // ERROR: z was not declared in this scope
    B::z = 3; // OK
}
```

Declarations

C++ code that introduces a name that can then be referred to is called *declaration*. There are many different kinds of declarations:

- variable declarations: `int a;`
- function declarations: `void foo();`
- namespace declarations: `namespace A { }`
- using declarations: `using A::x;`
- class declarations: `class C;`
- template declarations: `template <typename T> void foo();`
- ...

Declaration Specifiers

Some declarations can also contain additional *specifiers*. The following lists shows a few common ones and where they can be used. We will see some more specifiers in future lectures.

`static` Can be used for variable and function declarations, affects the declaration's *linkage* (see next slide). Also, objects declared with `static` have static storage duration.

`extern` Can be used for variable declarations in which case it also affects their linkage. Objects declared with `extern` also have static storage duration.

`inline` Can be used for variable and function declarations. Despite the name, has (almost) nothing to do with the inlining optimization. See the slides about the "One Definition Rule" for more information.

Linkage

Most declarations have a (conceptual) property called *linkage*. This property determines how the name of the declaration will be visible in the current and in other translation units. There are three types of linkage:

no linkage:

- Names can only be referenced from the scope they are in
- Local variables

internal linkage:

- Names can only be referenced from the same translation unit
- Global functions and variables declared with `static`
- Global variables that are not declared with `extern`
- All declarations in namespaces without name (“anonymous namespaces”)

external linkage:

- Names can be referenced from other translation units
- Global functions (without `static`)
- Global variables with `extern`

Definitions

When a name is declared it can be referenced by other code. However, most uses of a name also require the name to be *defined* in addition to be declared. Formally, this is called *odr-use* and covers the following cases:

- The value of a variable declaration is read or written
- The address of a variable or function declaration is taken
- A function is called
- An object of a class declaration is used

Most declarations are also definitions, with some exceptions such as

- Any declaration with an `extern` specifier and no initializer
- Function declarations without function bodies
- Declaration of a class name (“forward declaration”)

One Definition Rule (1)

One Definition Rule (ODR)

- At most one definition of a name is allowed *within one translation unit*
- Exactly one definition of every non-inline function or variable that is odr-used must appear *within the entire program*
- Exactly one definition of an inline-function must appear *within each translation unit* where it is odr-used
- Exactly one definition of a class must appear *within each translation unit* where the class is used and required to be complete

For subtleties and exceptions to these rules: See reference documentation

One Definition Rule (2)

a.cpp

```
int i = 5;  // OK: declares and defines i
int i = 6;  // ERROR: redefinition of i

extern int j; // OK: declares j
int j = 7;    // OK: (re-)declares and defines j
```

Separate declaration and definition is required to break circular dependencies

b.cpp

```
void bar();  // declares bar
void foo() { // declares and defines foo
    bar();
}
void bar() { // (re-)declares and defines bar
    foo();
}
```

One Definition Rule (3)

a.cpp

```cpp
int foo() {
    return 1;
}
```

b.cpp

```cpp
int foo() {
    return 2;
}
```

Trying to link a program consisting of a.cpp and b.cpp will fail

```
$ g++ -c -o a.o a.cpp
$ g++ -c -o b.o b.cpp
$ g++ a.o b.o
/usr/bin/ld: b.o: in function `foo()':
b.cpp:(.text+0x0): multiple definition of `foo()'; a.o:a.cpp:(.text+0x0): first
↪ defined here
collect2: error: ld returned 1 exit status
```

One Definition Rule (4)

What about helper functions/variables local to translation units? → Internal linkage!

- Option A: Use `static` (only works for variables and functions)

a.cpp
```
static int foo = 1;
static int bar() {
    return foo;
}
```

- Option B: Use *anonymous namespaces*

b.cpp
```
namespace {
//----------------------------------
int foo = 1;
int bar() {
    return foo;
}
//----------------------------------
}
```

Header and Implementation Files (1)

When distributing code over several files it is usually split into *header* and *implementation* files

- Header and implementation files have the same name, but different suffixes (e.g. `.hpp` for headers, `.cpp` for implementation files)
- Header files contain only declarations that should be visible and usable in other parts of the program
- Implementation files contain definitions of the names declared in the corresponding header
- At least the header files should include some documentation

Header and Implementation Files (2)

Why do we separate headers and implementation files?

- A `.cpp` file usually uses "external" functions and variables that are defined in another translation unit
- To compile a translation unit to an object file, the compiler needs to know the *declarations* of the external names
- Often it does not need to know the *definitions*
- Interdependent `.cpp` files can be compiled independently and simultaneously
- When only the definition and not the declaration of a function changes, no other translation units have to be recompiled
- Conceptual separation between "API" (in header files) and "Implementation" (in implementation files)

Note: In some cases the compiler does need the full definition of a name
→ Have to put definitions in headers in that case.

Header Guards (1)

A file may transitively include the same header multiple times

- May lead to unintentional redefinitions
- It is infeasible (and often impossible) to avoid duplicating transitive includes entirely
- Instead: Header files themselves ensure that they are included at most once in a single translation unit

path/A.hpp
```
inline int foo() { return 1; }
```

path/B.hpp
```
#include "path/A.hpp"
inline int bar() { return foo(); }
```

main.cpp
```
#include "path/A.hpp"
#include "path/B.hpp"  // ERROR: foo is defined twice
```

Header Guards (2)

Solution: Use header guards

path/A.hpp
```
// use any unique name, usually composed from the path
#ifndef H_path_A
#define H_path_A
inline int foo() { return 1; }
#endif
```

path/B.hpp
```
#ifndef H_path_B
#define H_path_B
#include "path/A.hpp"
inline int bar() { return foo(); }
#endif
```

Most compilers also support the non-standard `#pragma once` preprocessor directive. We recommend: Always use header guards.

Example: Header and Implementation Files (1)

The example CMake project from last lecture shows how header and implementation files are used. These are the header files:

sayhello.hpp

```
#ifndef H_exampleproject_sayhello
#define H_exampleproject_sayhello
#include <string_view>
/// Print a greeting for `name`
void sayhello(std::string_view name);
#endif
```

saybye.hpp

```
#ifndef H_exampleproject_saybye
#define H_exampleproject_saybye
#include <string_view>
/// Say bye to `name`
void saybye(std::string_view name);
#endif
```

Example: Header and Implementation Files (2)

The two header files have the following associated implementation files:

sayhello.cpp

```
#include "sayhello.hpp"
#include <iostream>

/// Print a greeting for `name`
void sayhello(std::string_view name) {
    std::cout << "Hello " << name << '!' << std::endl;
}
```

saybye.cpp

```
#include "saybye.hpp"
#include <iostream>

/// Say bye to `name`
void saybye(std::string_view name) {
    std::cout << "Bye " << name << '!' << std::endl;
}
```

Example: Header and Implementation Files (3)

The “main” file, in the example `print_greetings.cpp` only includes the headers:

print_greetings.cpp

```
#include <iostream>
#include "sayhello.hpp"
#include "saybye.hpp"

int main(int argc, const char** argv) {
    if (argc != 2) {
        std::cerr << "Please write: ./print_greetings name"
         ↪  << std::endl;
        return 1;
    }
    sayhello(argv[1]);
    saybye(argv[1]);
    return 0;
}
```

References, Arrays, and Pointers

Overview

So far, we have mostly worked with *fundamental types*

- `void`
- Arithmetic types such as `int`, `float`, etc.

Much of the power of C++ comes from the ability to define *compound types*

- Functions
- Classes (covered in the next lecture)
- References
- Arrays
- Pointers

Reference Declaration (1)

A reference declaration declares an alias to an already-existing object or function

- **Lvalue reference**: `type& declarator`
- **Rvalue reference**: `type&& declarator`
- Most of the time, *declarator* will simply be a name

References have some peculiarities

- There are no references to `void`
- References are immutable (although the referenced object may be mutable)
- References are not objects, i.e. they do not necessarily occupy storage

Since references are not objects

- There are no references or pointers to references
- There are no arrays of references

Reference Declaration (2)

The & or && tokens are part of the *declarator*, not the type

```
int i = 10;
int& j = i, k = i;  // j is reference to int, k is int
```

However, we may omit or insert whitespaces before and after the & or && tokens

- Both `int& j = i;` and `int &j = i;` are valid C++
- By convention, we use the former notation (`int& j = i;`)
- To avoid confusion, statements should declare only one identifier at a time
- Very rarely, exceptions to this rule are necessary in the *init-statements* of `if` and `switch` statements as well as `for` loops

Reference Initialization

Definitions of references to a type `T` must be initialized to refer to a valid object or function

- An object of type `T`
- A function of type `T`
- An object implicitly convertible to `T`

Declarations of references do not need intializers:

- Function parameter declarations
- Function return type declarations
- Class member declarations
- With the `extern` specifier

Lvalue References (1)

As an alias for existing objects

```
unsigned i = 10;
unsigned j = 42;
unsigned& r = i;  // r is an alias for i

r = 21;           // modifies i to be 21
r = j;            // modifies i to be 42

i = 123;
j = r;            // modifies j to be 123
```

Lvalue References (2)

To implement pass-by-reference semantics for function calls

```
void foo(int& value) {
    value += 42;
}

int main() {
    int i = 10;
    foo(i);     // i == 52
    foo(i);     // i == 94
}
```

Lvalue References (3)

To turn a function call into an lvalue expression

```
int global0 = 0;
int global1 = 0;

int& foo(unsigned which) {
    if (!which)
        return global0;
    else
        return global1;
}

int main() {
    foo(0) = 42;  // global0 == 42
    foo(1) = 14;  // global1 == 14
}
```

Rvalue References (1)

Can not (directly) bind to lvalues

```
int i = 10;
int&& j = i;  // ERROR: Cannot bind rvalue reference to lvalue
int&& k = 42; // OK
```

Extend the lifetimes of temporary objects

```
int i = 10;
int j = 32;

int&& k = i + j;  // k == 42
k += 42;          // k == 84;
```

Rvalue References (2)

Allow overload resolution to distinguish between lvalues and rvalues

```
void foo(int& x);
void foo(const int& x);
void foo(int&& x);

int& bar();
int baz();

int main() {
    int i = 42;
    const int j = 84;

    foo(i);      // calls foo(int&)
    foo(j);      // calls foo(const int&)
    foo(123);    // calls foo(int&&)

    foo(bar())  // calls foo(int&)
    foo(baz())  // calls foo(int&&)
}
```

References and CV-Qualifiers

References themselves cannot be cv-qualified

- However, the referenced type may be cv-qualified
- A reference to T can be initialized from a type that is less cv-qualified than T (e.g. `const int&` can be initialized from `int&`)

```
int i = 10;
const int& j = i;
int& k = j;  // ERROR: binding reference of type int& to
             //        const int discards cv-qualifiers
j = 42;      // ERROR: assignment of read-only reference
```

Lvalue references to `const` also extend the lifetime of temporary objects

```
int i = 10;
int j = 32;
const int& k = i + j;  // OK, but k is immutable
```

Dangling references

It is possible to write programs where the lifetime of a referenced object ends while references to it still exist.

- This can already happen when referencing objects with automatic storage duration
- Results in *dangling reference* and undefined behavior

Example

```
int& foo() {
    int i = 42;
    return i;    // ERROR: Returns dangling reference
}
```

Array Declaration (1)

An array declaration declares an object of array type (also: C-style array)

- `type declarator[expression]`
- `expression` must be an expression which evaluates to an integral constant at **compile time**
- Again, due to weird parsing rules `[expression]` is part of the declarator
- `type[expression]` can be used as a type outside of declarators

For example: `T a[N];` for some type `T` and compile-time constant `N`

- `a` consists of `N` contiguously allocated elements of type `T`
- Elements are numbered `0`, …, `N - 1`
- Elements can be accessed with the subscript operator `[]`, e.g. `a[0]`, …, `a[N - 1]`
- Without an initializer, every element of `a` is uninitialized

Array Declaration (2)

Example

```
unsigned short a[10];

for (unsigned i = 0; i < 10; ++i)
    a[i] = i + 1;
```

Array objects are lvalues, but they cannot be assigned to

```
unsigned short a[10];
unsigned short b[10];

a = b;  // ERROR: a is an array
```

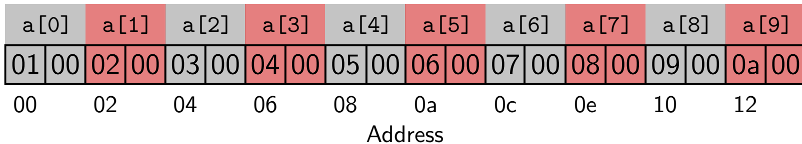
Arrays cannot be returned from functions

```
int[] foo();  // ERROR
```

Array Declaration (3)

Elements of an array are allocated **contiguously** in memory

- Given `unsigned short a[10];` containing the integers 1 through 10
- Assuming a 2-byte `unsigned short` type
- Assuming little-endian byte ordering



Arrays are just dumb chunks of memory

- Out-of-bounds accesses are not automatically detected, and do not necessarily lead to a crash
- May lead to rather weird bugs
- Exist mainly due to compatibility requirements with C

Array Declaration (4)

The elements of an array can be arrays themselves

```
unsigned short b[3][2];

for (unsigned i = 0; i < 3; ++i)
    for (unsigned j = 0; j < 2; ++j)
        b[i][j] = 3 * i + j;
```

Elements are still allocated contiguously in memory

- b can be thought of as a 3×2 matrix in row-major format
- The subscript operator simply returns an array object on the first level, to which the subscript operator can be applied again (`(b[i])[j]`)

Array Initialization

Arrays can be default-initialized, in which case every element is default-initialized

```
unsigned short a[10] = {};  // a contains 10 zeros
```

Arrays can be list-initialized, in which case the size may be omitted

```
unsigned short a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Multi-dimensional arrays may also be list-initialized, but only the first dimension may have unknown bound

```
unsigned short b[][2] = {
    {0, 1},
    {2, 3},
    {4, 5}
}
```

size_t

C++ has a designated type for indexes and sizes: `std::size_t` from `<cstddef>`

- `size_t` is an unsigned integer type that is large enough to represent sizes and all possible array indexes on the target architecture
- The C++ language and standard library use `size_t` when handling indexes or sizes
- Generally, use `size_t` for array indexes and sizes
- Sometimes you can also use smaller integer types (e.g. `unsigned`) when working with small arrays

std::array

C-style arrays should be avoided whenever possible

- Use the `std::array` type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking and other useful features
- `std::array` is a *template type* with two template parameters (the element type and count)

Example

```
#include <array>

int main() {
    std::array<unsigned short, 10> a;
    for (size_t i = 0; i < a.size(); ++i)
        a[i] = i + 1;  // no bounds checking
}
```

`std::vector` (1)

`std::array` is inflexible due to compile-time fixed size

- The `std::vector` type defined in the `<vector>` standard header provides dynamically-sized arrays
- Storage is automatically expanded and contracted as needed
- Elements are still stored contiguously in memory

Useful functions

- `push_back` – inserts an element at the end of the vector
- `size` – queries the current size
- `clear` – clears the contents
- `resize` – change the number of stored elements
- The subscript operator can be used with similar semantics as for C-style arrays

Familiarize yourself with the reference documentation on `std::vector`

std::vector (2)

Example

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

int main() {
    std::vector<unsigned short> a;
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    std::cout << a.size() << std::endl;  // prints 10
    a.clear();
    std::cout << a.size() << std::endl;  // prints 0

    a.resize(10);  // a now contains 10 zeros
    std::cout << a.size() << std::endl;  // prints 10

    for (unsigned i = 0; i < 10; ++i)
        a[i] = i + 1;
}
```

Range-For (1)

Execute a `for`-loop over a range

```
for (init-statement; range-declaration : range-expression)
    loop-statement
```

Explanation

- Executes *init-statement* once, then executes *loop-statement* once for each element in the range defined by *range-expression*
- *range-expression* may be an expression that represents a sequence (e.g. an array or an object for which `begin` and `end` functions are defined, such as `std::vector`)
- *range-declaration* should declare a named variable of the element type of the sequence, or a reference to that type
- *init-statement* may be omitted

Range-For (2)

Example

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

int main() {
    std::vector<unsigned short> a;

    // no range-for, we need the index
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    // range-for
    for (const unsigned short& e : a)
        std::cout << e << std::endl;
}
```

Storage of Objects

A "region of storage" has a physical equivalent

- Typically, objects reside in main memory, either on the stack or on the heap
- Up to now (and for some lectures to come), we have almost exclusively dealt with objects residing on the stack

Objects reside at some specific *location* in main memory

- As we have seen in the first lecture, this location can be identified by an *address* in main memory
- It is convenient to think of addresses as simple offsets from the beginning of the address space
- Pointers are a feature of C++ to obtain and interact with these addresses

Pointer Declaration (1)

A pointer declaration declares a variable of pointer type

- *type* cv declarator*
- *declarator* may be any other declarator, except for a reference declarator
- *cv* specifies the cv-qualifiers of the pointer (not the pointed-to type), and may be omitted
- Analogous to reference declarations, the * token is part of the declarator, not the type

Notes

- A pointer to an object represents the address of the *first* byte in memory that is occupied by that object
- As opposed to references, pointers are themselves objects
- Consequently, pointers to pointers are allowed

Pointer Declaration (2)

Examples of valid pointer declarations

```
int* a;                 // pointer to int
const int* b;           // pointer to const int
int* const c;           // const pointer to int
const int* const d;     // const pointer to const int
```

Pointer-to-pointer declarations

```
int** e;                      // pointer to pointer to int
const int* const* const f;    // const pointer to const pointer
                              // to const int
```

Contraptions like the declaration of `f` are very rarely (if at all) necessary

The Address-Of Operator

In general, there exists no implicit conversion from a pointed-to type (e.g. `int`) to its pointer type (e.g. `int*`)

- In order to obtain a pointer to an object, the built-in unary address-of operator & has to be used
- Given an lvalue expression a, &a returns a pointer to the value of the expression
- The cv-qualification of a is retained

Example

```
int a = 10;
const int b = 42;
int* c = &a;        // OK: c points to a
const int* d = &b;  // OK: d points to b
int* e = &b;        // ERROR: invalid conversion from
                    // const int* to int*
```

The Indirection Operator

In general, there exists no implicit conversion from a pointer type (e.g. `int*`) to its pointed-to type (e.g. `int`)

- In order to access the pointed-to object, the built-in unary indirection operator `*` has to be used
- Given an expression *expr* of pointer type, `*`*expr* returns an lvalue reference to the pointed-to object
- The cv-qualifiers of the pointed-to type are retained
- Applying the indirection operator is also called *dereferencing* a pointer

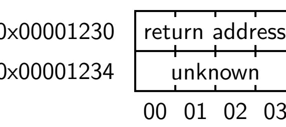
Example

```
int a = 10;
int* c = &a;
int& d = *c;  // reference to a
d = 123;      // a == 123
*c = 42;      // a == 42
```

What is Happening? (1)

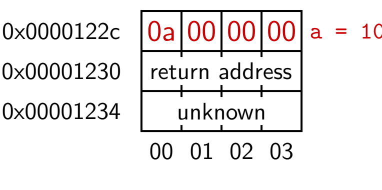
```
int main() {

}
```

Stack Memory



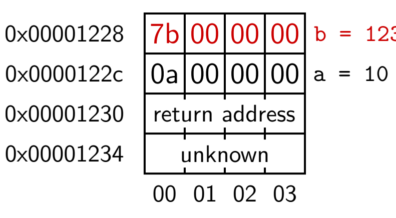
What is Happening? (2)

```
int main() {
    int a = 10;



}
```



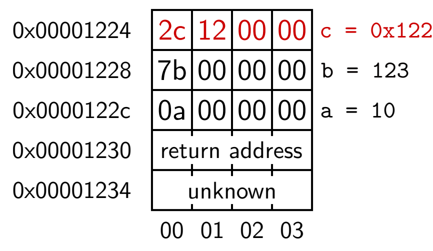
What is Happening? (3)

```
int main() {
    int a = 10;
    int b = 123;







}
```



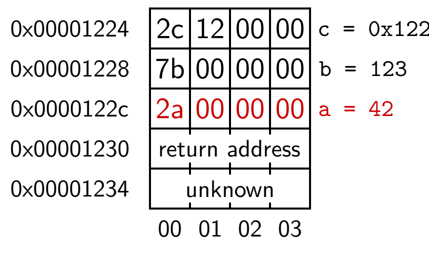
What is Happening? (4)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;


}
```



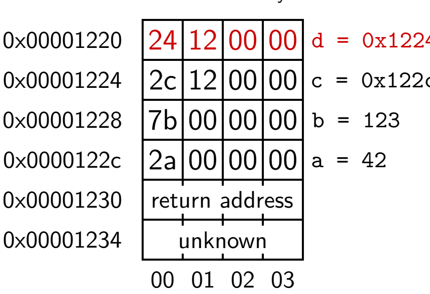
What is Happening? (5)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;



}
```



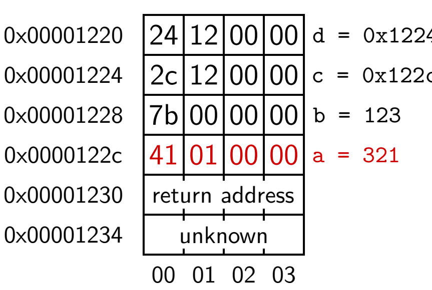
What is Happening? (6)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;


}
```



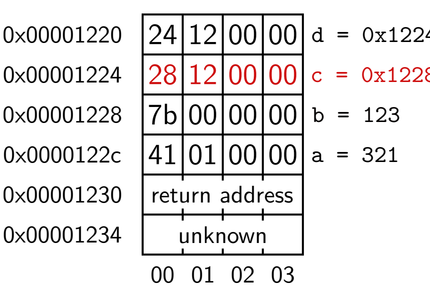
What is Happening? (7)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;



}
```



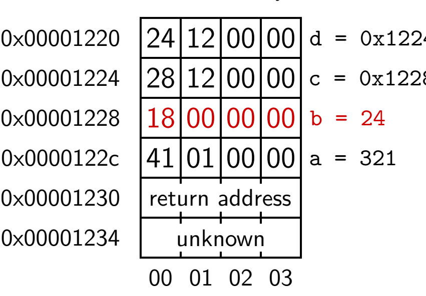
What is Happening? (8)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;


}
```



What is Happening? (9)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;


}
```



What is Happening? (10)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;

    return 0;
}
```

Stack Memory

0x00001234	unknown
	00 01 02 03

Null Pointers

A pointer may not point to any object at all

- Indicated by the special value and corresponding literal `nullptr`
- Pointers of the same type which are both null pointers are considered equal
- It is undefined behavior to dereference a null pointer

Undefined behavior can lead to surprising results

foo.cpp

```
int foo(const int* ptr) {
    int v = *ptr;

    if (ptr == nullptr)
        return 42;

    return v;
}
```

foo.o

```
foo(int*):
    movl    (%rdi), %eax
    ret
```

Array to Pointer Decay

Arrays and pointers have many similarities

- There is an implicit conversion from values of array type to values of pointer type
- The conversion constructs a pointer to the first element of an array
- The pointer type must be at least as cv-qualified as the array type

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The first element of array is ";
    std::cout << *ptr << std::endl;
}
```

The Subscript Operator

The subscript operator is defined on pointer types

- Treats the pointer as a pointer to the first element of an array
- Follows the same semantics as the subscript operator on array types

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The elements of array are";
    for (unsigned i = 0; i < 3; ++i)
        std::cout << " " << ptr[i];
    std::cout << std::endl;
}
```

Special Case: String Literals

String literals are another artifact of C compatibility

- String literals are immutable null-terminated character arrays
- That is, the type of a string literal with `N` characters is `const char[N + 1]`
- Most of the time, programmers take advantage of array-to-pointer decay and write `const char* str = "foo";`
- The character type can be controlled by the prefixes known from character literals (i.e. `u8"string"`, `u"string"`, or `U"string"`)

C-style strings should never be used!

- The C++ standard library provides the much safer `std::string` and `std::string_view` types
- Unfortunately, libraries or syscalls often require C-style string parameters
- If required, the standard library types can expose the C-style string representation

Arithmetic on Pointers (1)

Some arithmetic operators are defined between pointers and integral types

- Treats the pointer as a pointer to some element of an array
- Adding `i` to a pointer moves the it `i` elements to the right
- Subtracting `i` from a pointer moves it `i` elements to the left
- In general, for a pointer `p` the expressions `p[i]` and `*(p + i)` are equivalent

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = &array[1];

    std::cout << "The previous element is ";
    std::cout << *(ptr - 1) << std::endl;
    std::cout << "The next element is ";
    std::cout << *(ptr + 1) << std::endl;
}
```

Arithmetic on Pointers (2)

Special care has to be taken to only dereference valid pointers

- Especially important since it is valid to take the *past-the-end* pointer of an array or `std::vector`

Example

```
int main() {
    std::vector<int> v;
    v.resize(10);

    const int* firstPtr = &v[0];  // OK: valid pointer
    const int* lastPtr = &v[10];  // OK: past-the-end pointer

    int last1 = *lastPtr;   // ERROR, might segfault
    int last2 = v[10];      // ERROR, might segfault
}
```

Arithmetic on Pointers (3)

Subtraction is defined between pointers

- Treats both pointers as pointers to some elements of an array
- Computes the number of elements between these two pointers

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr1 = &array[0];
    const int* ptr2 = &array[3];  // past-the-end pointer

    std::cout << "There are " << (ptr2 - ptr1) << " elements ";
    std::cout << "in array" << std::endl;
}
```

Comparisons on Pointers

The comparison operators are defined between pointers

- Interprets the addresses represented by the pointers as integers and compares them
- Only defined if the pointers point to elements of the same array

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};

    std::cout << "The elements of array are"
    for (const int* it = &array[0]; it < &array[3]; ++it)
        std::cout << " " << *it;
    std::cout << std::endl;
}
```

Void Pointers

Pointers to `void` are allowed

- A pointer to an object of any type can implicitly be converted to a pointer to `void`
- The void pointer must be at least as cv-qualified as the original pointer
- The pointer value (i.e. the address) is unchanged

Usage

- Used to pass objects of unknown type
- Extensively used in C interfaces (e.g. `malloc`, `qsort`, ...)
- Only few operations are defined on void pointers (mainly assignment)
- In order to use the pointed-to object, one must *cast* the void pointer to the required type

static_cast (1)

The `static_cast` conversion is used to cast between related types

```
static_cast< new_type > ( expression )
```

Explanation

- Converts the value of *expression* to a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Can be used to convert void pointers to pointers of another type
- Many more use cases (see reference documentation)

static_cast (2)

Void pointers

```
int i = 42;
void* vp = &i;
int* ip = static_cast<int*>(vp);
```

Other related types

```
int sum(int a, int b);
double sum(double a, double b);

int main() {
    int a = 42;
    double b = 3.14;

    double x = sum(a, b);                       // ERROR: ambiguous
    double y = sum(static_cast<double>(a), b);  // OK
    int z = sum(a, static_cast<int>(b));        // OK
}
```

reinterpret_cast

The `reinterpret_cast` conversion is used to convert between unrelated types

```
reinterpret_cast < new_type > ( expression )
```

Explanation

- Interprets the underlying bit pattern of the value of *expression* as a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Usually does not generate any CPU instructions

Only a very restricted set of conversions is allowed

- A pointer to an object can be converted to a pointer to `std::byte`, `char` or `unsigned char`
- A pointer can be converted to an integral type (typically `uintptr_t`)
- Invalid conversions usually lead to **undefined behavior**

Strict Aliasing Rule (1)

It is **undefined behavior** to access an object using an expression of different type

- In particular, we are not allowed to access an object through a pointer to another type (pointer aliasing)
- Consequently, compilers typically assume that pointers to different types cannot have the same value
- There are very few exceptions to this rule

Strict Aliasing Rule (2)

foo.cpp

```cpp
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with `g++ -O1` will result in the following assembly

foo.o

```
main:
movl    $0, %eax
ret
```

Strict Aliasing Rule (3)

foo.cpp

```
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with `g++ -O2` will result in the following assembly

foo.o

```
main:
movl    $42, %eax
ret
```

Examining the Object Representation (1)

Important exception to the strict aliasing rule

- Any pointer may legally be converted to a pointer to `char`, or `unsigned char`
- Any pointer may legally be converted to a pointer to `std::byte` (defined in `<cstddef>` header, requires C++17),
- Permits the examination of the *object representation* of any object as an array of bytes

`std::byte` behaves similarly to `unsigned char`

- Represents a raw byte without any integer or character semantics
- Only bitwise operators are defined on bytes

Examining the Object Representation (2)

Example (compile with `g++ -std=c++20`)

```
#include <iostream>
#include <iomanip>
#include <cstddef>

int main() {
    double a = 3.14;
    const std::byte* bytes = reinterpret_cast<const std::byte*>(&a);

    std::cout << "The object representation of 3.14 is 0x";
    std::cout << std::hex << std::setfill('0') << std::setw(2);

    for (unsigned i = 0; i < sizeof(double); ++i)
        std::cout << static_cast<unsigned>(bytes[i]);

    std::cout << std::endl;
}
```

uintptr_t

Any pointer may legally be converted to an integral type

- The integral type must be large enough to hold all values of the pointer
- Usually, `uintptr_t` should be used (defined in `<cstdint>` header)
- Useful in some cases, especially when building custom data structures

Example

```
#include <cstddint>
#include <iostream>

int main() {
    int x = 42;
    uintptr_t addr = reinterpret_cast<uintptr_t>(&x);

    std::cout << "The address of x is " << addr << std::endl;
}
```

The `sizeof` Operator (1)

The `sizeof` operator queries the size of the object representation of a type

```
sizeof( type )
```

Explanation

- The size of a type is given in bytes
- `sizeof(std::byte)`, `sizeof(char)`, and `sizeof(unsigned char)` return `1` by definition
- Depending on the computer architecture, there may be 8 or more bits in one byte (as defined by C++)

The sizeof Operator (2)

The size of an object and pointer arithmetics are closely related

foo.cpp

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};

    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;

    int* ptr0 = &array[0];
    int* ptr1 = &array[1];

    uintptr_t uptr0 = reinterpret_cast<uintptr_t>(ptr0);
    uintptr_t uptr1 = reinterpret_cast<uintptr_t>(ptr1);

    std::cout << "(ptr1 - ptr0) = " << (ptr1 - ptr0) << std::endl;
    std::cout << "(uptr1 - uptr0) = " << (uptr1 - uptr0) << std::endl;
}
```

The `sizeof` Operator (3)

On an x86-64 machine, the program might produce the following output

```
$ ./foo
sizeof(int) = 4
(ptr1 - ptr0) = 1
(uptr1 - uptr0) = 4
```

Interpretation

- One `int` occupies 4 bytes
- There is one `int` between `ptr0` and `ptr1`
- There are 4 bytes (i.e. exactly one `int`) between `ptr0` and `ptr1`

The `alignof` Operator

Queries the alignment requirements of a type

```
alignof( type )
```

Explanation

- Depending on the computer architecture, certain types must have addresses aligned to specific byte boundaries
- The `alignof` operator returns the number of bytes between successive addresses where an object of *type* can be allocated
- The alignment requirement of a type is always a power of two
- Important (e.g.) for SIMD instructions, where the programmer must explicitly ensure correct alignment
- Memory accesses with incorrect alignment leads to undefined behavior, e.g. SIGSEGV or SIGBUS (depending on architecture)

Usage Guidelines

When to use references

- Pass-by-reference function call semantics
- When it is guaranteed that the referenced object will always be valid
- When object that should be referenced is always the same

When to use pointers

- Only when absolutely necessary!
- When there may not be a pointed-to object (i.e. `nullptr`)
- When the pointer may change to a different object
- When pointer arithmetic is desired

We will revisit this discussion later during the lecture

- Decision is intricately related to *ownership* semantics
- We would actually like to avoid using raw pointers as much as possible
- There are standard library classes which encapsulate pointers

Troubleshooting

Pointers have a reputation of being highly error-prone

- It is very easy to obtain pointers that point to invalid locations
- Once such a pointer is dereferenced, a number of bad things can happen

Bad things that may happen

- The pointer pointed outside of the program's address space
  - The program will likely segfault immediately
- The pointer pointed outside of the intended memory region, but still inside the program's address space
  - The program might segfault immediately
  - ...or simply corrupt some memory, which might lead to problems later

With the right tools, debugging is not as daunting as it may seem

The Infamous Segfault (1)

Every C++ programmer will encounter a segfault eventually

- Raised by hardware in response to a memory access violation
- In most cases caused by invalid pointers or memory corruption

Obvious example

foo.cpp

```
int main() {
    int* a;
    return *a;  // ERROR: Dereferencing an uninitialized pointer
}
```

Executing this program might result in the following

```
$ ./foo
[1]    5128 segmentation fault (core dumped)   ./foo
```

The Infamous Segfault (2)

Sometimes, the root cause may be (much) more difficult to determine

bar.cpp
```
int main() {
    long* ptr;
    long array[3] = {123, 456, 789};
    ptr = &array[0];
    array[3] = 987;  // ERROR: off-by-one access

    return *ptr;
}
```

When compiled with `g++ -fno-stack-protector`, this will also segfault

- The off-by-one access `array[3] = 987` actually changes the value of `ptr`
- Dereferencing this pointer in the return statement will result in a segfault
- The `-fno-stack-protector` option is required, because `g++` will by default emit extra code to prevent such buffer overflows

The Infamous Segfault (3)

Use the address sanitizer!

```
$ g++ -g -fno-stack-protector -obar bar.cpp
$ ./bar
[1]    4199 segmentation fault (core dumped)  ./bar
$ g++ -g -fno-stack-protector -fsanitize=address -obar bar.cpp
$ ./bar
=================================================================
==4229==ERROR: AddressSanitizer: stack-buffer-overflow on address [...]
WRITE of size 8 at 0x7fff536479d8 thread T0
    #0 0x5617976d529f in main (/tmp/bar+0x129f)
    #1 0x7f6a0fcc3022 in __libc_start_main (/usr/lib/libc.so.6+0x27022)
    #2 0x5617976d50ad in _start (/tmp/bar+0x10ad)

Address 0x7fff536479d8 is located in stack of thread T0 at offset 56 in
↪   frame
    #0 0x5617976d5188 in main (/tmp/bar+0x1188)

    This frame has 1 object(s):
      [32, 56) 'array' (line 3) <== Memory access at offset 56 overflows
↪   this variable
[...]
==4229==ABORTING
```

Classes

Classes

In C++ classes are the main kind of user-defined type.
Informal specification of a class definition:

```
class-keyword name {
    member-specification
};
```

- *class-keyword* is either `struct` or `class`
- *name* can be any valid identifier (like for variables, functions, etc.)
- *member-specification* is a list of declarations, mainly variables ("data members"), functions ("member functions"), and types ("nested types")
- The trailing semicolon is mandatory!

Data Members

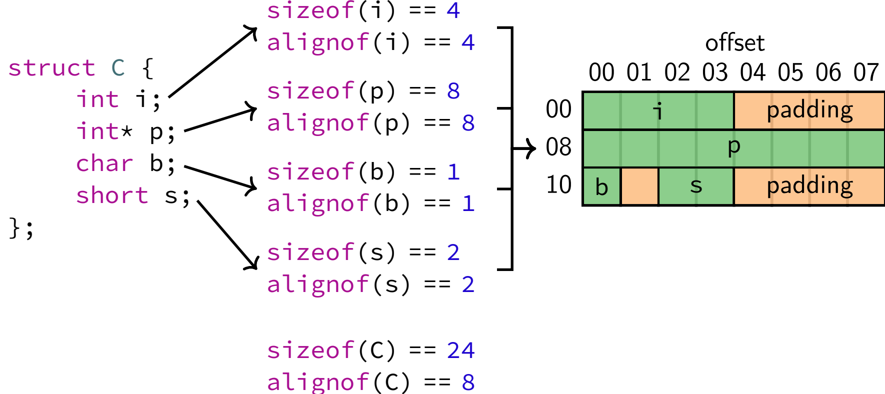
- Declarations of data members are variable declarations
- `extern` is not allowed
- Declarations without `static` are called *non-static* data members, otherwise they are *static* data members
- `thread_local` is only allowed for static data members
- Declaration must have a *complete type* (see later slide)
- Name of the declaration must differ from the class name and must be unique within the class
- Non-static data members can have a *default value*

```
struct Foo {
    // non-static data members:
    int a = 123;
    float& b;
    const char c;
    // static data members:
    static int s;
    thread_local static int t;
};
```

Memory Layout of Data Members

- Every type has a size and an alignment requirement
- To be compatible between different compilers and programming languages (mainly C), the memory layout of objects of class type is fixed
- Non-static data members appear in memory by the order of their declarations
- Size and alignment of each data-member is accounted for → leads to "gaps" in the object, called *padding bytes*
- Alignment of a class type is equal to the largest alignment of all non-static data members
- Size of a class type is at least the sum of all sizes of all non-static data members and at least 1
- static data members are stored separately

Size, Alignment and Padding



Reordering the member variables in the order p, i, s, b would lead to `sizeof(C) == 16`!
In general: Order member variables by decreasing alignment to get the fewest padding bytes.

Member Functions

- Declarations of member functions are like regular function declarations
- Just like for data members, there are non-static and static (with the `static` specifier) member functions
- Non-static member functions can be *const-qualified* (with `const`) or *ref-qualified* (with `const&`, `&`, or `&&`)
- Non-static member functions can be `virtual`
- There are some member functions with special functions:
    - Constructor and destructor
    - Overloaded operators

```
struct Foo {
    void foo(); // non-static member function
    void cfoo() const; // const-qualified non-static member function
    void rfoo() &; // ref-qualified non-static member function
    static void bar(); // static member function
    Foo(); // Constructor
    ~Foo(); // Destructor
    bool operator==(const Foo& f); // Overloaded operator ==
};
```

Accessing Members

Given the following code:

```
struct C {
    int i;
    static int si;
};
C o; // o is variable of type C
C* p = &o; // p is pointer to o
```

the members of the object can be accessed as follows:

- non-static and static member variables and functions can be accessed with the *member-of* operator: `o.i`, `o.si`
- As a shorthand, instead of writing `(*p).i`, it is possible to write `p->i`
- Static member variables and functions can also be accessed with the *scope resolution* operator: `C::si`

Writing Member Functions

- In a non-static member function members can be accessed implicitly without using the member-of operator (preferred)
- Every non-static member function has the implicit parameter `this`
- In member functions without qualifiers and ref-qualified ones `this` has the type `C*`
- In const-qualified or const-ref-qualified member functions `this` has the type `const C*`

```
struct C {
    int i;
    int foo() {
        this->i; // Explicit member access, this has type C*
        return i; // Implicit member access
    }
    int foo() const { return this->i; /* this has type const C* */ }
    int bar() & { return i; /* this (implicit) has type C* */ }
    int bar() const& { return this->i; /* this has type const C* */ }
};
```

Out-of-line Definitions

- Just like regular functions member functions can have separate declarations and definitions
- A member function that is defined in the class body is said to have an *inline definition*
- A member function that is defined outside of the class body is said to have an *out-of-line definition*
- Member functions with inline definitions implicitly have the `inline` specifier
- Out-of-line definitions must have the same qualifiers as their declaration

```
struct Foo {
    void foo1() { /* ... */ } // Inline definition
    void foo2();
    void foo_const() const;
    static void foo_static();
};
// Out-of-line definitions
void Foo::foo2() { /* ... */ }
void Foo::foo_const() const { /* ... */ }
void Foo::foo_static() { /* ... */ }
```

Forward Declarations (1)

Classes can be *forward-declared*

- Syntax: `class-keyword name ;`
- Declares a class type which will be defined later in the scope
- The class name has *incomplete type* until it is defined
- The forward-declared class name may still be used in some situations (more details next)

Use Cases

- Allows classes to refer to each other
- Can reduce compilation time (significantly) by avoiding transitive includes of an expensive-to-compile header
- Commonly used in header files

Forward Declarations (2)

Example

foo.hpp

```
class A;
class ClassFromExpensiveHeader;

class B {
    ClassFromExpensiveHeader* member;

    void foo(A& a);
};
class A {
    void foo(B& b);
};
```

foo.cpp

```
#include "expensive_header.hpp"

/* implementation */
```

Incomplete Types

A forward-declared class type is *incomplete* until it is defined

- In general, no operations that require the size and layout of a type to be known can be performed on an incomplete type
  - E.g. pointer arithmetics on a pointer to an incomplete type
  - E.g. Definition or call (but not declaration) of a function with incomplete return or argument type
- However, some declarations can involve incomplete types
  - E.g. pointer declarations to incomplete types
  - E.g. member function declarations with incomplete parameter types
- For details: See the reference documentation

Constructors

- Constructors are special functions that are called when an object is *initialized*
- Constructors have no return type, no const- or ref-qualifiers, and their name is equal to the class name
- The definition of a constructor can have an *initializer list*
- Constructors can have arguments, a constructor without arguments is called *default constructor*
- Constructors are sometimes implicitly defined by the compiler

```
struct Foo {
    Foo() {
        std::cout << "Hello\n";
    }
};
```

```
struct Foo {
    int a;
    Bar b;
    // Default constructor is
    // implicitly defined, does
    // nothing with a, calls
    // default constructor of b
};
```

Initializer List

- The initializer list specifies how member variables are initialized before the body of the constructor is executed
- Other constructors can be called in the initializer list
- Members should be initialized in the order of their definition
- Members are initialized to their default value if not specified in the list
- `const` member variables can only be initialized in the initializer list

```
struct Foo {
    int a = 123; float b; const char c;
    // default constructor initializes a (to 123), b, and c
    Foo() : b(2.5), c(7) {}
    // initializes a and b to the given values
    Foo(int a, float b, char c) : a(a), b(b), c(c) {}
    Foo(float f) : Foo() {
        // First the default constructor is called, then the body
        // of this constructor is executed
        b *= f;
    }
};
```

Initializing Objects

- When an object of class type is initialized, an appropriate constructor is executed
- Arguments given in the initialization are passed to the constructor
- C++ has several types of initialization that are very similar but unfortunately have subtle differences:
  - *default initialization* (`Foo f;`)
  - *value initialization* (`Foo f{};` and `Foo()`)
  - *direct initialization* (`Foo f(1, 2, 3);`)
  - *list initialization* (`Foo f{1, 2, 3};`)
  - *copy initialization* (`Foo f = g;`)
- Simplified syntax: `class-type identifier(arguments);` or `class-type identifier{arguments};`

Converting and Explicit Constructors

- Constructors with exactly one argument are treated specially: They are used for *explicit* and *implicit conversions*
- If implicit conversion with such constructors is not desired, the keyword `explicit` can be used to disallow it
- Generally, you should use `explicit` unless you have a good reason not to

```
struct Foo {
    Foo(int i);
};
void print_foo(Foo f);
// Implicit conversion,
// calls Foo::Foo(int)
print_foo(123);
// Explicit conversion,
// calls Foo::Foo(int)
static_cast<Foo>(123);
```

```
struct Bar {
    explicit Bar(int i);
};
void print_bar(Bar f);
// Implicit conversion,
// compiler error!
print_bar(123);
// Explicit conversion,
// calls Bar::Bar(int)
static_cast<Bar>(123);
```

Copy Constructors

- Constructors of a class `C` that have a single argument of type `C&` or `const C&` (preferred) are called *copy constructors*
- They are often called implicitly by the compiler whenever it is necessary to copy an object
- The copy constructor if often implicitly defined by the compiler

```
struct Foo {
    Foo(const Foo& other) { /* ... */ }
};
void doFoo(Foo f);
Foo f;
Foo g(f); // Call copy constructor explicitly
doFoo(g); // Copy constructor is called implicitly
```

Destructors

- The destructor is a special function that is called when the lifetime of an object ends
- The destructor has no return type, no arguments, no const- or ref-qualifiers, and its name is `~`*`class-name`*
- For objects with automatic storage duration (e.g. local variables) the destructor is called implicitly at the end of the scope in reverse order of their definition

```
Foo a;
Bar b;
{
    Baz c;
    // c.~Baz() is called;
}
// b.~Bar() is called
// a.~Foo() is called
```

Writing Destructors

- The destructor is a regular function that can contain any code
- Most of the time the destructor is used to explicitly free resources
- Destructors of member variables are called automatically at the end in reverse order

```
struct Foo {
    Bar a;
    Bar b;
    ~Foo() {
        std::cout << "Bye\n";
        // b.~Bar() is called
        // a.~Bar() is called
    }
};
```

Member Access Control

- Every member of a class has `public`, `protected`, or `private` access
- When the class is defined with `class`, the default access is `private`
- When the class is defined with `struct`, the default access is `public`
- `public` members can be accessed by everyone, `protected` members only by the class itself and its subclasses, `private` members only by the class itself

```
class Foo {
    int a; // a is private
    public:
    // All following declarations are public
    int b;
    int getA() const { return a; }
    protected:
    // All following declarations are protected
    int c;
    public:
    // All following declarations are public
    static int getX() { return 123; }
};
```

Friend Declarations (1)

A class body can contain *friend declarations*

- A friend declaration grants a function or another class access to the private and protected members of the class which contains the declaration
- Syntax: `friend` *function-declaration* `;`
  - Declares a function as a friend of the class
- Syntax: `friend` *function-definition* `;`
  - Defines a non-member function and declares it as a friend of the class
- Syntax: `friend` *class-specifier* `;`
  - Declares another class as a friend of this class

Notes

- Friendship is non-transitive and cannot be inherited
- Access specifiers have no influence on friend declarations (i.e. they can appear in `private:` or `public:` sections)

Friend Declarations (2)

Example

```
class A {
    int a;
    friend class B;
    friend void foo(A&);
};
class B {
    friend class C;
    void bar(A& a) {
        a.a = 42;  // OK
    }
};
class C {
    void foo(A& a) {
        a.a = 42;  // ERROR
    }
};
void foo(A& a) {
    a.a = 42;  // OK
}
```

Nested Types

- For nested types classes behave just like a namespace
- Nested types are accessed with the scope resolution operator ::
- Nested types are friends of their parent

```
struct A {
    struct B {
        int getI(const A& a) {
            return a.i; // OK, B is friend of A
        }
    };
    private:
    int i;
};
A::B b; // reference nested type B of class A
```

Constness of Member Variables

- Accessing a member variable through a *non-const lvalue* yields a *non-const lvalue* if the member is non-const and a *const lvalue* otherwise
- Accessing a member variable through a *const lvalue* yields a *const lvalue*
- Exception: Member variables declared with `mutable` yield a *non-const lvalue* even when accessed through a *const lvalue*

```
struct Foo {
    int i;
    const int c;
    mutable int m;
}
Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;
```

Expression	Value Category
`foo.i`	non-const lvalue
`foo.c`	const lvalue
`foo.m`	non-const lvalue
`cfoo.i`	const lvalue
`cfoo.c`	const lvalue
`cfoo.m`	non-const lvalue

Constness and Member Functions

- The value category through which a non-static member function is accessed is taken into account for overload resolution
- For *non-const lvalues* non-const overloads are preferred over const ones
- For *const lvalues* only const-(ref-)qualified functions are selected

```
struct Foo {
  int getA() { return 1; }
  int getA() const { return 2; }
  int getB() & { return getA(); }
  int getB() const& { return getA(); }
  int getC() const { return getA(); }
  int getD() { return 3; }
};
Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;
```

Expression	Value
`foo.getA()`	1
`foo.getB()`	1
`foo.getC()`	2
`foo.getD()`	3
`cfoo.getA()`	2
`cfoo.getB()`	2
`cfoo.getC()`	2
`cfoo.getD()`	*error*

Casting and CV-qualifiers

- When using `static_cast`, `reinterpret_cast`, or `dynamic_cast`, cv-qualifiers cannot be "casted away"
- `const_cast` must be used instead
- Syntax: `const_cast < new_type > ( expression )`
- *new_type* may be a pointer or reference to a class type
- *expression* and *new_type* must have same type ignoring their cv-qualifiers
- The result of `const_cast` is a value of type *new_type*
- Modifying a const object through a non-const access path is undefined behavior!

```
struct Foo {
    int a;
};
const Foo f{123};
Foo& fref = const_cast<Foo&>(f); // OK, cast is allowed
int b = fref.a; // OK, accessing value is allowed
fref.a = 42; // undefined behavior
```

Use Cases for const_cast

Most common use case of `const_cast`: Avoid code duplication in member function overloads.

- A class may contain a const and non-const overload of the same function with identical code
- Should only be used when absolutely necessary (i.e. not for simple overloads)

```
class A {
    int* numbers;
    int& foo() {
        int i = /* ... */;
        // do some incredibly complicated computation to
        // get a value for i
        return numbers[i]
    }
    const int& foo() const {
        // OK as long as foo() does not modify the object
        return const_cast<A&>(*this).foo();
    }
};
```

Operator Overloading

- Classes can have special member functions to overload built-in operators like `+`, `==`, etc.
- Many overloaded operators can also be written as non-member functions
- Syntax: *return-type* `operator` *op* (*arguments*)
- Overloaded operator functions are selected with the regular overload resolution
- Overloaded operators are not required to have meaningful semantics
- Almost all operators can be overloaded, exceptions are: `::` (scope resolution), `.` (member access), `.*` (member pointer access), `?:` (ternary operator)
- This includes "unusual" operators like: `=` (assignment), `()` (call), `*` (dereference), `&` (address-of), `,` (comma)

Arithmetic Operators

The expression *lhs op rhs* is mostly equivalent to *lhs*.`operator` *op*(*rhs*) or `operator` *op*(*lhs*, *rhs*) for binary operators.

- As calls to overloaded operators are treated like regular function calls, the overloaded versions of || and && lose their special behaviors
- Should be const and take const references
- Usually return a value and not a reference
- The unary + and − operators can be overloaded as well

```
struct Int {
    int i;
    Int operator+(const Int& other) const { return Int{i + other.i}; }
    Int operator-() const { return Int{-i}; };
};
Int operator*(const Int& a, const Int& b) { return Int{a.i * b.i}; }

Int a{123}; Int b{456};

a + b; /* is equivalent to */ a.operator+(b);
a * b; /* is equivalent to */ operator*(a, b);
-a; /* is equivalent to */ a.operator-();
```

Comparison Operators

All binary comparison operators (<, <=, >, >=, ==, !=, <=>) can be overloaded.

- Should be const and take const references
- Return `bool`, except for <=> (see next slide)
- If only `operator<=>` is implemented, <, <=, >, and >= work as well
- `operator==` must be implemented separately
- If `operator==` is implemented, != works as well

```
struct Int {
    int i;
    std::strong_ordering operator<=>(const Int& a) const {
        return i <=> a.i;
    }
    bool operator==(const Int& a) const { return i == a.i; }
};
Int a{123}; Int b{456};
a < b; /* is equivalent to */ (a.operator<=>(b)) < 0;
a == b; /* is equivalent to */ a.operator==(b);
```

Three-Way Comparison (1)

The overloaded `operator<=>` should return one of the following three types from `<compare>`: `std::partial_ordering`, `std::weak_ordering`, `std::strong_ordering`.

- When comparing two values a and b with `ord = (a <=> b)`, then `ord` has one of the three types and can be compared to 0:
- `ord == 0` ⇔ `a == b`
- `ord < 0` ⇔ `a < b`
- `ord > 0` ⇔ `a > b`
- `std::strong_ordering` can be converted to `std::weak_ordering` and `std::partial_ordering`
- `std::weak_ordering` can be converted to `std::partial_ordering`

Three-Way Comparison (2)

`std::partial_ordering` should be used when two values can potentially be unordered, i.e. `a <= b` and `a >= b` could be false.
Possible values:

- `std::partial_ordering::less`
- `std::partial_ordering::equivalent`
- `std::partial_ordering::greater`
- `std::partial_ordering::unordered`

Three-Way Comparison (3)

`std::weak_ordering` or `std::strong_ordering` should be used when two values are always ordered (i.e. we have *total order*).
Possible values:

- `std::weak_ordering::less`
- `std::weak_ordering::equivalent`
- `std::weak_ordering::greater`
- `std::strong_ordering::less`
- `std::strong_ordering::equivalent`
- `std::strong_ordering::greater`
- With `std::strong_odering` equal values must also be “indistinguishable”, i.e. behave the same in all aspects

Increment and Decrement Operators

Overloaded pre- and post-increment and -decrement operators are distinguished by an (unused) `int` argument.

- `C& operator++(); C& operator--();` overloads the pre-increment or -decrement operator, usually modifies the object and then returns `*this`
- `C operator++(int); C operator--(int);` overloads the post-increment or -decrement operator, usually copies the object before modifying it and then returns the unmodified copy

```
struct Int {
    int i;
    Int& operator++() { ++i; return *this; }
    Int operator--(int) { Int copy{*this}; --i; return copy; }
};
Int a{123};
++a; // a.i is now 124
a++; // ERROR: post-increment is not overloaded
Int b = a--; // b.i is 124, a.i is 123
--b; // ERROR: pre-decrement is not overloaded
```

Subscript Operator

Classes that behave like containers or pointers usually override the *subscript operator* `[]`.

- `a[b]` is equivalent to `a.operator[](b)`
- Type of `b` can be anything, for array-like containers it is usually `size_t`

```
struct Foo { /* ... */ };
struct FooContainer {
    Foo* fooArray;
    Foo& operator[](size_t n) { return fooArray[n]; }
    const Foo& operator[](size_t n) const { return fooArray[n]; }
};
```

Dereference Operators

Classes that behave like pointers usually override the operators * (dereference) and -> (member of pointer).

- `operator*()` usually returns a reference
- `operator->()` should return a pointer or an object that itself has an overloaded -> operator

```
struct Foo { /* ... */ };
struct FooPtr {
    Foo* ptr;
    Foo& operator*() { return *ptr; }
    const Foo& operator*() const { return *ptr; }
    Foo* operator->() { return ptr; }
    const Foo* operator->() const { return ptr; }
};
```

Assignment Operators

- The simple assignment operator is often used together with the copy constructor and should have the same semantics
- All assignment operators usually return `*this`

```
struct Int {
    int i;
    Foo& operator=(const Foo& other) { i = other.i; return *this; }
    Foo& operator+=(const Foo& other) { i += other.i; return *this; }
};
Foo a{123};
a = Foo{456}; // a.i is now 456
a += Foo{1}; // a.i is now 457
```

Conversion Operators

A class `C` can use converting constructors to convert values of other types to type `C`. Similarly, *conversion operators* can be used to convert objects of type `C` to other types.

Syntax: `operator` *type* `()`

- Conversion operators have the implicit return type *type*
- They are usually declared as `const`
- The `explicit` keyword can be used to prevent implicit conversions
- Explicit conversions are done with `static_cast`
- `operator bool()` is usually overloaded to be able to use objects in an `if` statement

```
struct Int {
    int i;
    operator int() const {
        return i;
    }
};
Int a{123};
int x = a; // OK, x is 123
```

```
struct Float {
    float f;
    explicit operator float() const {
        return f;
    }
};
Float b{1.0};
float y = b; // ERROR, implicit conversion
float y = static_cast<float>(b); // OK
```

Argument-Dependent Lookup

- Overloaded operators are usually defined in the same namespace as the type of one of their arguments
- Regular unqualified lookup would not allow the following example to compile
- To fix this, unqualified names of functions are also looked up in the *namespaces of all arguments*
- This is called *Argument Dependent Lookup (ADL)*

```
namespace A { class X {}; X operator+(const X&, const X&); }
int main() {
    A::X x, y;
    operator+(x, y); // Need operator+ from namespace A
    A::operator+(x, y); // OK
    x + y; // How to specify namespace here?
           // -> ADL finds A::operator+()
}
```

Defaulted Member Functions

- Most of the time the implementation of default constructors, copy constructors, copy assignment operators, and destructors is trivial
- To let the compiler generate the trivial implementation automatically, `= default;` can be used instead of a function body

```
struct Foo {
    Bar b;
    Foo() = default; /* equivalent to: */ Foo() {}
    ~Foo() = default; /* equivalent to: */ ~Foo() {}

    Foo(const Foo& f) = default;
    /* equivalent to: */
    Foo(const Foo& f) : b(f.b) {}

    Foo& operator=(const Foo& f) = default;
    /* equivalent to: */
    Foo& operator=(const Foo& f) {
        b = f.b; return *this;
    }
};
```

Defaulted Comparison Operators

All comparison operators can be defaulted.

- Defaulted comparison operators must return `bool`, except `<=>`
- Defaulted `operator==` compares each member for equality, members must define `operator==`
- Defaulted `operator<=>` lexicographically compares members by using `<=>`, members must define `operator<=>`
- Defaulting `operator<=>` also defaults `operator==`
- Defaulted `<`, `<=`, `>`, or `>=` use `operator<=>`

```
struct Int128 {
    int64_t x; int64_t y;
    std::strong_ordering operator<=>(const Int&) const = default;
};
Int128 a{0, 123}; Int128 b{1, 0};
a < b; // true
a == b; // false
a <=> b; // std::strong_ordering::less
```

Deleted Member Functions

- Sometimes, implicitly generated constructors or assignment operators are not wanted
- Writing `= delete;` instead of a function body explicitly forbids implicit definitions
- In other cases the compiler implicitly deletes a constructor in which case writing `= default;` enables it again

```
struct Foo {
    Foo(const Foo&) = delete;
};
Foo f; // Default constructor is defined implicitly
Foo g(f); // ERROR: copy constructor is deleted
```

Other User-Defined Types

Unions

- In addition to regular classes declared with `class` or `struct`, there is another special class type declared with `union`
- In a union only one member may be "active", all members use the same storage
- Size of the union is equal to size of largest member
- Alignment of the union is equal to largest alignment among members
- Strict aliasing rule still applies with unions!
- Most of the time there are better alternatives to unions, e.g. `std::array<char, N>` or `std::variant`

```
union Foo {
    int a;
    double b;
};
sizeof(Foo) == 8;
alignof(Foo) == 8;
```

```
Foo f; // No member is active
f.a = 1; // a is active
std::cout << f.b; // Undefined behavior!
f.b = 12.34; // Now, b is active
std::cout << f.b; // OK
```

Enums

- C++ also has user-defined enumeration types
- Typically used like integral types with a restricted range of values
- Also used to be able to use descriptive names instead of “magic” integer values
- Syntax: *enum-key name* { *enum-list* };
- *enum-key* can be `enum`, `enum class`, or `enum struct`
- *enum-list* consists of comma-separated entries with the following syntax: *name* [= *value*]
- When *value* is not specified, it is automatically chosen starting from 0

```
enum Color {
    Red, // Red == 0
    Blue, // Blue == 1
    Green, // Green == 2
    White = 10,
    Black, // Black == 11
    Transparent = White // Transparent == 10
};
```

Using Enum Values

- Names from the enum list can be accessed with the scope resolution operator
- When `enum` is used as keyword, names are also introduced in the enclosing namespace
- Enums declared with `enum` can be converted implicitly to `int`
- Enums can be converted to integers and vice versa with `static_cast`
- `enum class` and `enum struct` are equivalent
- Guideline: Use `enum class` unless you have a good reason not to

```
Color::Red; // Access with scope resolution operator
Blue; // Access from enclosing namespace
int i = Color::Green; // i == 2, implicit conversion
int j = static_cast<int>(Color::White); // j == 10
Color c = static_cast<Color>(11); // c == Color::Black
```

Type Aliases

- Names of types that are nested deeply in multiple namespaces or classes can become very long
- Sometimes it is useful to declare a nested type that refers to another, existing type
- For this *type aliases* can be used
- Syntax: `using` *`name`* `=` *`type`*`;`
- *`name`* is the name of the alias, *`type`* must be an existing type
- For compatibility with C type aliases can also be defined with `typedef` with a different syntax but this should never be used in modern C++ code

```
namespace A::B::C { struct D { struct E {}; }; }
using E = A::B::C::D::E;
E e; // e has type A::B::C::D::E
struct MyContainer {
    using value_type = int;
};
MyContainer::value_type i = 123; // i is an int
```

Common Type Aliases

In C++ the following aliases are defined in the `std` namespace and are commonly used:

- `intN_t`: Integer types with exactly `N` bits, usually defined for 8, 16, 32, and 64 bits
- `uintN_t`: Similar to `intN_t` but unsigned
- `size_t`: Used by the standard library containers everywhere a size or index is needed, also result type of `sizeof` and `alignof`
- `uintptr_t`: An integer type that is guaranteed to be able to hold all possible values that result from a `reinterpret_cast` from any pointer
- `intptr_t`: Similar to `uintptr_t` but signed
- `ptrdiff_t`: Result type of expressions that subtract two pointers
- `max_align_t`: Type which has alignment as least as large as all other scalar types

Dynamic Memory Management

Process Memory Layout (1)

Each Linux process runs within its own virtual address space

- The kernel pretends that each process has access to a (huge) continuous range of addresses ($\approx$ 256 TiB on x86-64)
- Virtual addresses are mapped to physical addresses by the kernel using page tables and the MMU (if available)
- Greatly simplifies memory management code in the kernel and improves security due to memory isolation
- Allows for useful “tricks” such as memory-mapping files

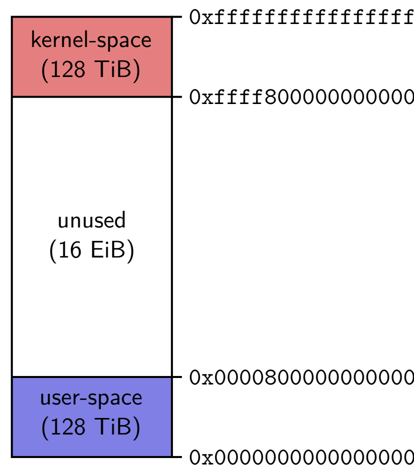
Process Memory Layout (2)

The kernel also uses virtual memory

- Part of the address space has to be reserved for kernel memory
- This kernel-space memory is mapped to the same physical addresses for each process
- Access to this memory is restricted

Most of the address space is unused

- MMUs on x86-64 platforms only support 48 bit pointers at the moment
- Might change in the future (Linux already supports 56 bit pointers)



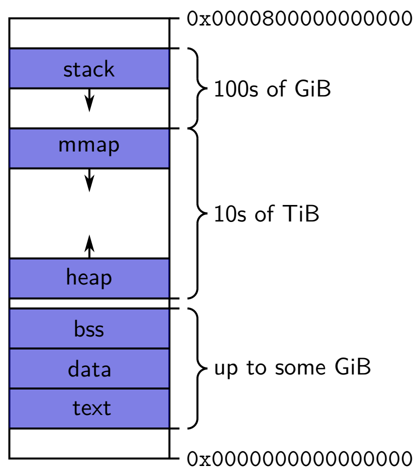
Process Memory Layout (3)

User-space memory is organized in segments

- Stack segment
- Memory mapping segment
- Heap segment
- BSS, data and text segments

Segments can grow

- Stack and memory mapping segments usually grow down (i.e. addresses decrease)
- Heap segment usually grows up (i.e. addresses increase)



Stack Segment (1)

Stack memory is typically used for objects with automatic storage duration

- The compiler can statically decide when allocations and deallocations must happen
- The memory layout is known at compile-time
- Allows for highly optimized code (allocations and deallocations simply increase/decrease a pointer)

Fast, but inflexible memory

- Array sizes must be known at compile-time
- No dynamic data structures are possible (trees, graphs, etc.)

Stack Segment (2)

Example

foo.cpp

```cpp
int foo() {
    int c = 2;
    int d = 21;

    return c * d;
}

int main() {
    int a[100];
    int b = foo();

    return b;
}
```

foo.o

```
foo():
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $2, -4(%rbp)
    movl    $21, -8(%rbp)
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    popq    %rbp
    ret
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $416, %rsp
    call    foo()
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    leave
    ret
```

Heap Segment

The heap is typically used for objects with dynamic storage duration

- The *programmer* must explicitly manage allocations and deallocations
- Allows much more flexible programs

Disadvantages

- Performance impact due to non-trivial implementation of heap-based memory allocation
- Memory fragmentation
- Dynamic memory allocation is error-prone
  - Memory leaks
  - Double free (deallocation)
  - Make use of debugging tools (GDB, ASAN (!))

Dynamic Memory Management in C++

C++ provides several mechanisms for dynamic memory management

- Through `new` and `delete` expressions (discouraged)
- Through the C functions `malloc` and `free` (discouraged)
- Through smart pointers and ownership semantics (preferred)

Mechanisms give control over the storage duration and possibly lifetime of objects

- Level of control varies by method
- In all cases: Manual intervention required

The new Expression

Creates and initializes objects with dynamic storage duration

- Syntax: `new` *`type initializer`*
- *`type`* must be a type
- *`type`* can be an array type
- *`initializer`* can be omitted

Explanation

- Allocates heap storage for a single object or an array of objects
- Constructs and initializes a single object or an array of objects in the newly allocated storage
- If *`initializer`* is absent, the object is default-initialized
- Returns a pointer to the object or the initial element of the array

The `delete` Expression

Every object allocated through `new` must be destroyed through `delete`

- Syntax (single object): `delete expression`
- *`expression`* must be a pointer created by the single-object form of the `new` expression
- Syntax (array): `delete[] expression`
- *`expression`* must be a pointer created by the array form of the `new` expression
- In both cases *`expression`* may be `nullptr`

Explanation

- If *`expression`* is `nullptr` nothing is done
- Invokes the destructor of the object that is being destroyed, or of every object in the array that is being destroyed
- Deallocates the memory previously occupied by the object(s)

new & delete Example

```
class IntList {
    struct Node {
        int value;
        Node* next;
    };

    Node* first;
    Node* last;

    public:
    ~IntList() {
        while (first != nullptr) {
            Node* next = first->next;
            delete first;
            first = next;
        }
    }

    void push_back(int i) {
        Node* node = new Node{i, nullptr};
        if (!last)
            first = node;
        else
            last->next = node;
        last = node;
    }
};
```

Memory Leaks

Memory leaks can happen easily

```
int foo(unsigned length) {
    int* buffer = new int[length];

    /* ... do something ... */

    if (condition)
        return 42;  // MEMORY LEAK

    /* ... do something else ... */

    delete[] buffer;
    return 123;
}
```

Avoid explicit memory management through `new` and `delete` whenever possible

Placement new (1)

Constructs objects in already allocated storage

- Syntax: `new (placement_params) type initializer`
- *placement_params* must be a pointer to a region of storage large enough to hold an object of type *type*
- The strict aliasing rule must not be violated
- Alignment must be ensured manually
- Only rarely required (e.g. for custom memory management)
- Requires that the <new> standard header is included

Placement new (2)

Example

```
#include <cstddef>
#include <new>

struct A { };

int main() {
        std::byte* buffer = new std::byte[sizeof(A)];
        A* a = new (buffer) A();
        /* ... do something with a ... */
        a->~A();  // we must explicitly call the destructor
        delete[] buffer;
}
```

Lifetimes and Storage Duration (1)

The lifetime of an object is equal to or nested within the lifetime of its storage

- Equal for regular `new` and `delete`
- Possibly nested for placement `new`

Example

```
struct A { };

int main() {
    A* a1 = new A();        // lifetime of a1 begins, storage begins
    a1->~A();               // lifetime of a1 ends
    A* a2 = new (a1) A();   // lifetime of a2 begins
    delete a2;              // lifetime of a2 ends, storage ends
}
```

Lifetimes and Storage Duration (2)

Lifetime and storage duration of objects have real-world implications

- Accessing objects outside of their lifetime is undefined behavior and will often lead to segmentation faults
- Important to always keep track of lifetimes (if necessary through suitable comments)
- Use debugging tools (in particular ASAN) to find such bugs!

Examples of common bugs

- Returning pointers/references to local variables from functions
- Using a pointer/reference to access memory that has already been freed
- Using a pointer/reference to access an object that has already been destructed
- Maintaining pointers/references to objects in an `std::vector` after its internal storage has been reallocated (e.g. through a call to push_back)
- ...

std::memcpy (1)

std::memcpy copies bytes between non-overlapping memory regions

- Defined in <cstring> standard header
- Syntax: `void* memcpy(void* dest, const void* src, std::size_t count);`
- Copies count bytes from the object pointed to by src to the object pointed to by dest
- Can be used to work around strict aliasing rules without causing undefined behavior

Restrictions (undefined behavior if violated)

- Objects must not overlap
- src and dest must not be nullptr
- Objects must be *trivially copyable*
- dest must be aligned suitably

std::memcpy (2)

Example (straightforward copy)

```
#include <cstring>
#include <vector>

int main() {
    std::vector<int> buffer = {1, 2, 3, 4};
    buffer.resize(8);
    std::memcpy(&buffer[4], &buffer[0], 4 * sizeof(int));
}
```

Example (work around strict aliasing)

```
#include <cstring>
#include <cstdint>

int main() {
    int64_t i = 42;
    double j;
    std::memcpy(&j, &i, sizeof(double)); // OK
}
```

std::memmove (1)

std::memmove copies bytes between possibly overlapping memory regions

- Defined in <cstring> standard header
- Syntax: `void* memmove(void* dest, const void* src, std::size_t count);`
- Copies count bytes from the object pointed to by src to the object pointed to by dest
- Acts as if the bytes were copied to a temporary buffer

Restrictions (undefined behavior if violated)

- src and dest must not be nullptr
- Objects must be *trivially copyable*
- dest must be suitably aligned

std::memmove (2)

Example (straightforward copy)

```
#include <cstring>
#include <vector>

int main() {
    std::vector<int> buffer = {1, 2, 3, 4};
    buffer.resize(6);
    std::memmove(&buffer[2], &buffer[0], 4 * sizeof(int));
    // buffer is now {1, 2, 1, 2, 3, 4}
}
```

Copy and Move Semantics

Copy Semantics

Assignment and construction of classes employs *copy semantics* in most cases

- By default, a shallow copy is created
- Usually not particularly relevant for fundamental types
- Very relevant for user-defined class types

Considerations for user-defined class types

- Copying may be expensive
- Copying may be unnecessary or even unwanted
- An object on the left-hand side of an assignment might manage dynamic resources

Copy Constructor (1)

Invoked whenever an object is initialized from an object of the same type

- Syntax: *class_name* `(` `const` *class_name*`&` `)`
- *class_name* must be the name of the current class

For a class type `T` and objects `a`, `b`, the copy constructor is invoked on

- Copy initialization: `T a = b;`
- Direct initialization: `T a(b);`
- Function argument passing: `f(a);` where `f` is `void f(T t);`
- Function return: `return a;` inside a function `T f();` if `T` has no move constructor (more details next)

Copy Constructor (2)

Example

```
class A {
    private:
    int v;

    public:
    explicit A(int v) : v(v) { }
    A(const A& other) : v(other.v) { }
};

int main() {
    A a1(42);    // calls A(int)

    A a2(a1);    // calls copy constructor
    A a3 = a2;   // calls copy constructor
}
```

Copy Assignment (1)

Typically invoked if an object appears on the left-hand side of an assignment with an lvalue on the right-hand side

- Syntax (1): `class_name& operator=( class_name )`
- Syntax (2): `class_name& operator=( const class_name& )`
- *`class_name`* must be the name of the current class
- Usually, option (2) is preferred unless the *copy-and-swap* idiom is used (more details next)

Explanation

- Called whenever selected by overload resolution
- Returns a reference to the object itself (i.e. `*this`) to allow for chaining assignments

Copy Assignment (2)

Example

```
class A {
    private:
    int v;

    public:
    explicit A(int v) : v(v) { }
    A(const A& other) : v(other.v) { }

    A& operator=(const A& other) {
        v = other.v;
        return *this;
    }
};

int main() {
    A a1(42);   // calls A(int)
    A a2 = a1;  // calls copy constructor

    a1 = a2;    // calls copy assignment operator
}
```

Implicit Declaration (1)

The compiler will implicitly declare a copy constructor if no user-defined copy constructor is provided

- The implicitly declared copy constructor will be a `public` member of the class
- The implicitly declared copy constructor may or may not be defined

The implicitly declared copy constructor is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be copy-constructed
- The class has a base class which cannot be copy-constructed
- The class has a base class with a deleted or inaccessible destructor
- The class has a user-defined move constructor or assignment operator
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the constructor.

Implicit Declaration (2)

The compiler will implicitly declare a copy assignment operator if no user-defined copy assignment operator is provided

- The implicitly declared copy assignment operator will be a `public` member of the class
- The implicitly declared copy assignment operator may or may not be defined

The implicitly declared copy assignment operator is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be copy-assigned
- The class has a base class which cannot be copy-assigned
- The class has a non-static data member of reference type
- The class has a user-defined move constructor or assignment operator
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the assignment operator.

Implicit Definition

If it is not deleted, the compiler defines the implicitly-declared copy constructor

- Only if it is actually used (odr-used)
- Performs a full member-wise copy of the object's bases and members in their initialization order
- Uses direct initialization

If it is not deleted, the compiler defines the implicitly-declared copy assignment operator

- Only if it is actually used (odr-used)
- Performs a full member-wise copy assignment of the object's bases and members in their initialization order
- Uses built-in assignment for scalar types and copy assignment for class types

Example: Implicit Declaration & Definition

Example

```
struct A {
    const int v;

    explicit A(int v) : v(v) { }
};

int main() {
    A a1(42);

    A a2(a1);  // OK: calls the generated copy constructor
    a1 = a2;   // ERROR: the implicitly-declared copy assignment
               //        operator is deleted
}
```

Trivial Copy Constructor and Assignment Operator (1)

The copy constructor/assignment operator may be *trivial*

- It must not be user-provided (explicitily defaulting does not count as user-provided)
- The class has no virtual member functions
- The copy constructor/assignment operator for all direct bases and non-static data members of class type is trivial

A trivial copy constructor/assignment operator behaves similar to `std::memcpy`

- Every scalar subobject is copied recursively and no further action is performed
- The object representation of the copied object is not necessarily identical to the source object
- Trivially copyable objects may legally be copied using `std::memcpy`
- All data types compatible with C are trivially copyable

Trivial Copy Constructor and Assignment Operator (2)

Example

```
#include <vector>

struct A {
    int b;
    double c;
};

int main() {
    std::vector<A> buffer1;
    buffer1.resize(10);

    std::vector<A> buffer2;     // copy buffer1 using copy-constructor
    for (const A& a : buffer1)
        buffer2.push_back(a);

    std::vector<A> buffer3;     // copy buffer1 using memcpy
    buffer3.resize(10);
    std::memcpy(&buffer3[0], &buffer1[0], 10 * sizeof(A));
}
```

Implementing Custom Copy Operations (1)

Custom copy constructors/assignment operators are only **occasionally** necessary

- Often, a class should not be copyable anyway if the implicitly generated versions do not make sense
- Exceptions include classes which manage some kind of resource (e.g. dynamic memory)

Guidelines for implementing custom copy operations

- The programmer should either provide neither a copy constructor nor a copy assignment operator, or both
- The copy assignment operator should usually include a check to detect self-assignment
- If possible, resources should be reused
- If resources cannot be reused, they have to be cleaned up properly

Implementing Custom Copy Operations (2)

Example

```
struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(const A& other) : A(other.capacity) {
        std::memcpy(memory, other.memory, capacity * sizeof(int));
    }
    ~A() { delete[] memory; }

    A& operator=(const A& other) {
        if (this == &other)                   // check for self-assignment
            return *this;

        if (capacity != other.capacity) {  // attempt to reuse resources
            delete[] memory;
            capacity = other.capacity;
            memory = new int[capacity];
        }

        std::memcpy(memory, other.memory, capacity * sizeof(int));

        return *this;
    }
};
```

Move Semantics

Copy semantics often incur unnecessary overhead or are unwanted

- An object may be immediately destroyed after it is copied
- An object might not want to share a resource it is holding

Move semantics provide a solution to such issues

- Move constructors/assignment operators typically “steal” the resources of the argument
- Leave the argument in a valid but indeterminate state
- Greatly enhances performance in some cases

Move Construction (1)

Typically called when an object is initialized from an rvalue of the same type

- Syntax: `class_name ( class_name&& ) noexcept`
- *`class_name`* must be the name of the current class
- The `noexcept` keyword should be added to indicate that the constructor never throws an exception

Explanation

- Overload resolution decides if the copy or move constructor of an object should be called
- Temporary values and calls to functions that return an object are rvalues
- The `std::move` function in the `<utility>` header may be used to convert an lvalue to an rvalue
- We know that the argument does not need its resources anymore, so we can simply steal them

Move Construction (2)

For a class type `T` and objects `a`, `b`, the move constructor is invoked on

- Direct initialization: `T a(std::move(b));`
- Copy initialization: `T a = std::move(b);`
- Function argument passing: `f(std::move(b));` with `void f(T t);`
- Function return: `return a;` inside `T f();`

Example

```
struct A {
    A(const A& other);
    A(A&& other);
};
A getA();
int main() {
    A a1;
    A a2(a1);             // calls copy constructor
    A a3(std::move(a1));  // calls move constructor
    A a4(getA());         // calls move constructor
}
```

Move Assignment (1)

Typically called if an object appears on the left-hand side of an assignment with an rvalue on the right-hand side

- Syntax: `class_name& operator=( class_name&& ) noexcept`
- *`class_name`* must be the name of the current class
- The `noexcept` keyword should be added to indicate that the assignment operator never throws an exception

Explanation

- Overload resolution decides if the copy or move assignment operator of an object should be called
- We know that the argument does not need its resources anymore, so we can simply steal them
- The move assignment operator returns a reference to the object itself (i.e. `*this`) to allow for chaining

Move Assignment (2)

Example

```
struct A {
    A();
    A(const A&);
    A(A&&) noexcept;

    A& operator=(const A&);
    A& operator=(A&&) noexcept;
};

int main() {
    A a1;
    A a2 = a1;              // calls copy-constructor
    A a3 = std::move(a1);   // calls move-constructor

    a3 = a2;                // calls copy-assignment
    a2 = std::move(a3);     // calls move-assignment
}
```

Implicit Declaration (1)

The compiler will implicitly declare a `public` move constructor if all the following conditions hold

- There are no user-declared copy constructors
- There are no user-declared copy assignment operators
- There are no user-declared move assignment operators
- There are no user-declared destructors

The implicitly declared move constructor is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be moved
- The class has a base class which cannot be moved
- The class has a base class with a deleted or inaccessible destructor
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the constructor.

Implicit Declaration (2)

The compiler will implicitly declare a `public` move assignment operator if all the following conditions hold

- There are no user-declared copy constructors
- There are no user-declared copy assignment operators
- There are no user-declared move constructors
- There are no user-declared destructors

The implicitly declared move assignment operator is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be moved
- The class has non-static data members of reference type
- The class has a base class which cannot be moved
- The class has a base class with a deleted or inaccessible destructor
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the assignment operator.

Implicit Definition

If it is not deleted, the compiler defines the implicitly-declared move constructor

- Only if it is actually used (odr-used)
- Performs a full member-wise move of the object's bases and members in their initialization order
- Uses direct initialization

If it is not deleted, the compiler defines the implicitly-declared move assignment operator

- Only if it is actually used (odr-used)
- Performs a full member-wise move assignment of the object's bases and members in their initialization order
- Uses built-in assignment for scalar types and move assignment for class types

Example: Implicit Declaration & Definition

Example

```
struct A {
    const int v;

    explicit A(int v) : v(v) { }
};

int main() {
    A a1(42);

    A a2(std::move(a1));  // OK: calls the generated move constructor
    a1 = std::move(a2);   // ERROR: the implicitly-declared move
                          //        assignment operator is deleted
}
```

Trivial Move Constructor and Assignment Operator

The move constructor/assignment operator may be *trivial*

- It must not be user-provided (explicitily defaulting does not count as user-provided)
- The class has no virtual member functions
- The move constructor/assignment operator for all direct bases and non-static data members of class type is trivial

A trivial move constructor/assignment operator acts similar to `std::memcpy`

- Every scalar subobject is copied recursively and no further action is performed
- The object representation of the copied object is not necessarily identical to the source object
- Trivially movable objects may legally be moved using `std::memcpy`
- All data types compatible with C are trivially movable

Implementing Custom Move Operations (1)

Custom move constructors/assignment operators are **often** necessary

- A class that manages some kind of resource *almost always* requires custom move constructors and assignment operators

Guidelines for implementing custom move operations

- The programmer should either provide neither a move constructor nor a move assignment operator, or both
- The move assignment operator should usually include a check to detect self-assignment
- The move operations should typically not allocate new resources, but steal the resources from the argument
- The move operations should leave the argument in a valid state
- Any previously held resources must be cleaned up properly

Implementing Custom Move Operations (2)

Example

```
struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(A&& other) noexcept : capacity(other.capacity), memory(other.memory) {
        other.capacity = 0;
        other.memory = nullptr;
    }
    ~A() { delete[] memory; }

    A& operator=(A&& other) noexcept {
        if (this == &other)          // check for self-assignment
            return *this;

        delete[] memory;
        capacity = other.capacity;
        memory = other.memory;

        other.capacity = 0;
        other.memory = nullptr;

        return *this;
    }
};
```

Copy Elision (1)

Compilers must omit copy and move constructors under certain circumstances

- Objects are instead directly constructed in the storage into which they would be copied/moved
- Results in zero-copy pass-by-value semantics
- Most importantly in return statements and variable initialization from a temporary
- More optimizations allowed, but not required

This is one of very few optimizations which is allowed to change observable side-effects

- Not all compilers perform the same optional optimizations
- Programs that rely on side-effects of copy/move constructors and destructors are not portable

Copy Elision (2)

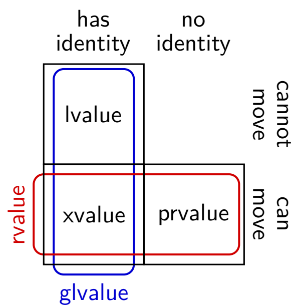
Example

```
#include <iostream>

struct A {
    int a;

    A(int a) : a(a) {
        std::cout << "constructed" << std::endl;
    }

    A(const A& other) : a(other.a) {
        std::cout << "copy-constructed" << std::endl;
    }
};

A foo() {
    return A(42);
}

int main() {
    A a = foo();  // prints only "constructed"
}
```

Value Categories

Move semantics and copy elision require a more sophisticated taxonomy of expressions

- glvalues identify objects
- xvalues identify an object whose resources can be reused
- prvalues compute the value of an operand or initialize an object



In particular, `std::move` just converts its argument to an xvalue expression

- `std::move` is exactly equivalent to a `static_cast` to an rvalue reference
- `std::move` is exclusively syntactic sugar (to guide overload resolution)

Copy-And-Swap (1)

The copy-and-swap idiom is convenient if copy assignment cannot benefit from resource reuse

- The class defines only the `class_type& operator=( class_type )` copy-and-swap assignment operator
- Acts both as copy and move assignment operator depending on the value category of the argument

Implementation

- Exchange the resources between the argument and `*this`;
- Let the destructor clean up the resources of the argument

Copy-And-Swap (2)

Example

```
#include <algorithm>
#include <cstring>

struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(const A& other) : A(other.capacity) {
        std::memcpy(memory, other.memory, capacity * sizeof(int));
    }
    ~A() { delete[] memory; }

    A& operator=(A other) { // copy/move constructor is called to create other
        std::swap(capacity, other.capacity);
        std::swap(memory, other.memory);

        return *this;
    }  // destructor cleans up resources formerly held by *this
};
```

Temporarily uses more resources than strictly required

The Rule of Three

If a class requires one of the following, it almost certainly requires all three

- A user-defined destructor
- A user-defined copy constructor
- A user-defined copy assignment operator

Explanation

- Having a user-defined copy constructor usually implies some custom setup logic which needs to be executed by copy assignment and vice-versa
- Having a user-defined destructor usually implies some custom cleanup logic which needs to be executed by copy assignment and vice-versa
- The implicitly-defined versions are usually incorrect if a class manages a resource of non-class type (e.g. a raw pointer, POSIX file descriptor, etc.)

The Rule of Five

If a class follows the rule of three, move operations are defined as deleted

- If move semantics are desired for a class, it has to define all five special member functions
- If only move semantics are desired for a class, it still has to define all five special member functions, but define the copy operations as deleted

Explanation

- Not adhering to the rule of five usually does not lead to incorrect code
- However, many optimization opportunities may be inaccessible to the compiler if no move operations are defined

Resource Acquisition is Initialization (1)

Bind the lifetime of a resource that has to be allocated to the lifetime of an object

- Resources can be allocated heap memory, sockets, files, mutexes, disk space, database connections, etc.
- Guarantees availability of the resource during the lifetime of the object
- Guarantees that resources are released when the lifetime of the object ends
- Object should have automatic storage duration
- Known as the **Resource Acquisition is Initialization (RAII)** idiom

One of the most important and powerful idioms in C++!

- One consequence: Never use `new` and `delete` outside of an RAII class
- C++ already defines *smart pointers* that are RAII wrappers for `new` and `delete`
- Thus we almost never need to use `new` and `delete` in our code

Resource Acquisition is Initialization (2)

Implementation of RAII

- Encapsulate each resource into a class whose sole responsibility is managing the resource
- The constructor acquires the resource and establishes all class invariants
- The destructor releases the resource
- Typically, copy operations should be deleted and custom move operations need to be implemented

Usage of RAII classes

- RAII classes should only be used with automatic or temporary storage duration
- Ensures that the compiler manages the lifetime of the RAII object and thus indirectly manages the lifetime of the resource

Resource Acquisition is Initialization (3)

Example

```
class CustomIntBuffer {
private:
    int* memory;
public:
    explicit CustomIntBuffer(unsigned size) : memory(new int[size]) { }
    CustomIntBuffer(const CustomIntBuffer&) = delete;
    CustomIntBuffer(CustomIntBuffer&& other) noexcept : memory(other.memory) {
        other.memory = nullptr;
    }
    ~CustomIntBuffer() { delete[] memory; }

    CustomIntBuffer& operator=(const CustomIntBuffer&) = delete;
    CustomIntBuffer& operator=(CustomIntBuffer&& other) noexcept {
        if (this != &other) {
            delete[] memory;
            memory = other.memory;
            other.memory = nullptr;
        }
        return *this;
    }

    int* getMemory() { return memory; }
    const int* getMemory() const { return memory; }
};
```

Resource Acquisition is Initialization (4)

Example usage of the `CustomIntBuffer` class

```
#include <utility>

bool foo(CustomIntBuffer buffer) {
    /* do something */

    if (condition)
        return false;  // no worries about forgetting to free memory

    /* do something more */

    return true;       // no worries about forgetting to free memory
}

int main() {
    CustomIntBuffer buffer(5);

    return foo(std::move(buffer));
}
```

Ownership

Ownership Semantics

One of the main challenges in manual memory management is tracking ownership

- Traditionally, owners can be, e.g., functions or classes
- Only the owner of some dynamically allocated memory may safely free it
- Multiple objects may have a pointer to the same dynamically allocated memory

The RAII idiom and move semantics together enable *ownership semantics*

- A resource should be "owned", i.e. encapsulated, by exactly one C++ object at all times
- Ownership can only be transferred explicitly by moving the respective object
- E.g., the `CustomIntBuffer` class implements ownership semantics for a dynamically allocated `int`-array

std::unique_ptr (1)

std::unique_ptr is a so-called *smart pointer*

- Essentially implements RAII/ownership semantics for arbitrary pointers
- Assumes unique ownership of another C++ object through a pointer
- Automatically disposes of that object when the std::unique_ptr goes out of scope
- A std::unique_ptr may own no object, in which case it is empty
- Can be used (almost) exactly like a raw pointer
- But: std::unique_ptr can only be moved, not copied

std::unique_ptr is defined in the <memory> standard header

- It is a template class, and can be used for arbitrary types
- Syntax: std::unique_ptr< type > where one would otherwise use type*

std::unique_ptr should *always* be preferred over raw pointers!

std::unique_ptr (2)

Usage of `std::unique_ptr` (for details: see reference documentation)

Creation

- `std::make_unique<type>(arg0, ..., argN)`, where `arg0, ..., argN` are passed to the constructor of `type`

Indirection, subscript, and member access

- The indirection, subscript, and member access operators `*`, `[]` and `->` can be used in the same way as for raw pointers

Conversion to bool

- `std::unique_ptr` is contextually convertible to `bool`, i.e. it can be used in `if` statements in the same way as raw pointers

Accessing the raw pointer

- The `get()` member function returns the raw pointer
- The `release()` member function returns the raw pointer and releases ownership

std::unique_ptr (3)

Example

```
#include <memory>

struct A {
    int a;
    int b;

    A(int a, int b) : a(a), b(b) { }
};
void foo(std::unique_ptr<A> aptr) {  // assumes ownership
    /* do something */
}
void bar(const A& a) { // does not assume ownership
    /* do something */
}
int main() {
    std::unique_ptr<A> aptr = std::make_unique<A>(42, 123);
    int a = aptr->a;
    bar(*aptr);              // retain ownership
    foo(std::move(aptr));   // transfer ownership
}
```

std::unique_ptr (4)

std::unique_ptr can also be used for heap-based arrays

```
std::unique_ptr<int[]> foo(unsigned size) {
    std::unique_ptr<int[]> buffer = std::make_unique<int[]>(size);

    for (unsigned i = 0; i < size; ++i)
        buffer[i] = i;

    return buffer;  // transfer ownership to caller
}

int main() {
    std::unique_ptr<int[]> buffer = foo(42);

    /* do something */
}
```

std::shared_ptr (1)

Rarely, true *shared ownership* is desired

- A resource may be simultaneously have several owners
- The resource should only be released once the last owner releases it
- `std::shared_ptr` defined in the `<memory>` standard header can be used for this
- Multiple `std::shared_ptr` objects may own the same raw pointer (implemented through reference counting)
- `std::shared_ptr` may be copied and moved

Usage of `std::shared_ptr`

- Use `std::make_shared` for creation
- Remaining operations analogous to `std::unique_ptr`
- For details: See the reference documentation

`std::shared_ptr` is rather expensive and should be avoided when possible

std::shared_ptr (2)

Example

```
#include <memory>
#include <vector>

struct Node {
    std::vector<std::shared_ptr<Node>> children;

    void addChild(std::shared_ptr<Node> child);
    void removeChild(unsigned index);
};

int main() {
    Node root;
    root.addChild(std::make_shared<Node>());
    root.addChild(std::make_shared<Node>());
    root.children[0]->addChild(root.children[1]);

    root.removeChild(1);  // does not free memory yet
    root.removeChild(0);  // frees memory of both children
}
```

Usage Guidelines: Pointers (1)

`std::unique_ptr` represents ownership

- Used for dynamically allocated objects
  - Frequently required for polymorphic objects
  - Useful to obtain a movable handle to an immovable object
- `std::unique_ptr` as a function parameter or return type indicates a transfer of ownership
- `std::unique_ptr` should almost always be passed *by value*

Raw pointers represent resources

- Should almost always be encapsulated in RAII classes (mostly `std::unique_ptr`)
- Very occasionally, raw pointers are desired as function parameters or return types
  - If ownership is not transferred, but there might be no object (i.e. `nullptr`)
  - If ownership is not transferred, but pointer arithmetic is required

Usage Guidelines: References (2)

References grant temporary access to an object without assuming ownership

- If necessary, a reference can be obtained from a smart pointer through the indirection operator `*`

Ownership can also be relevant for other types (e.g. `std::vector`)

- Moving (i.e. transferring ownership) should always be preferred over copying
- Should be passed *by lvalue-reference* if ownership is not transferred
- Should be passed *by rvalue-reference* if ownership is transferred
- Should be passed *by value* if they should be copied

Rules can be relaxed if an object is not copyable

- Should be passed *by lvalue-reference* if ownership is not transferred
- Should be passed *by value* if ownership is transferred

Usage Guidelines (3)

Example

```
struct A { };

// reads a without assuming ownership
void readA(const A& a);
// may read and modify a but doesn't assme ownership
void readWriteA(A& a);
// assumes ownership of A
void consumeA(A&& a);
// works on a copy of A
void workOnCopyOfA(A a);

int main() {
    A a;

    readA(a);
    readWriteA(a);
    workOnCopyOfA(a);
    consumeA(std::move(a));  // cannot call without std::move
}
```

Usage Guidelines: Function Arguments (1)

When dealing with an object of type T use the following rough guidelines to decide which type to use when passing it as function argument:

Situation	Type to Use
▪ Ownership of object should be transferred to callee ▪ Potential copies are acceptable or T is not copy-able ▪ Object is relatively small (at most ≈ one cache line)	`T`
▪ Ownership of object should be transferred to callee ▪ Object is relatively large (more than ≈ one cache line), so it should live on the heap	`std::unique_ptr<T>`

Usage Guidelines: Function Arguments (2)

Situation	Type to Use
• Ownership of object should *not* be transferred to callee • Callee should not modify object • Object is larger than a pointer	`const T&`
• Ownership of object should *not* be transferred to callee • Callee is expected to modify the object	`T&`
• Same as `const T&`, but should be nullable	`const T*`
• Same as `T&`, but should be nullable	`T*`

Inheritance

Object-Oriented Programming

Object-oriented programming is based on three fundamental concepts

- Data abstraction
  - Implemented by classes in C++
  - Covered previously
- Inheritance
  - Implemented by class derivation in C++
  - Derived Classes inherit the members of its base class(es)
  - Covered in this lecture
- Dynamic Binding (Polymorphism)
  - Implemented by virtual functions in C++
  - Programs need not care about the specific types of objects in an inheritance hierarchy
  - Covered in this lecture

Derived Classes (1)

Any class type may be derived from one or more *base classes*

- Possible for both `class` and `struct`
- Base classes may in turn be derived from their own base classes
- Classes form an *inheritance hierarchy*

High-level Syntax

```
class class-name : base-specifier-list {
    member-specification
};
```

```
struct class-name : base-specifier-list {
    member-specification
};
```

Derived Classes (2)

The *base-specifier-list* contains a comma-separated list of one or more *base-specifier*s with the following syntax

```
access-specifier virtual-specifier base-class-name
```

Explanation

- *access-specifier* controls the *inheritance mode* (more details soon)
- *access-specifier* is optional; if present it can be one of the keywords `private`, `protected` or `public`
- *base-class-name* is mandatory, it specifies the name of the class from which to derive
- *virtual-specifier* is optional; if present it must be the keyword `virtual` (only used for multiple inheritance)

Derived Classes (3)

Examples

```
class Base {
    int a;
};

class Derived0 : Base {
    int b;
};

class Derived1 : private Base {
    int c;
};

class Derived2 : public virtual Base, private Derived1 {
    int d;
};
```

Constructors and Initialization (1)

Constructors of derived classes account for the inheritance

1. The direct non-virtual base classes are initialized in left-to-right order as they appear in the `base-specifier-list`
2. The non-static data members are initialized in the order of declaration in the class definition
3. The body of the constructor is executed

The initialization order is independent of any order in the member initializer list

Base classes are default-initialized unless specified otherwise

- Another constructor can explicitly be invoked using the delegating constructor syntax

Constructors and Initialization (2)

Consider the class definitions

foo.hpp

```
struct Base {
    int a;

    Base();
    explicit Base(int a);
};

struct Derived : Base  {
    int b;

    Derived();
    Derived(int a, int b);
};
```

foo.cpp

```
#include "foo.hpp"
#include <iostream>

using namespace std;

Base::Base()
    : a(42) {
    cout << "Base::Base()" << endl;
}

Base::Base(int a)
    : a(a) {
    cout << "Base::Base(int)" << endl;
}

Derived::Derived() {
    : b(42) {
    cout << "Derived::Derived()" << endl;
}

Derived::Derived(int a, int b)
    : Base(a), b(b) {
    cout << "Derived::Derived(int, int)" << endl;
}
```

Constructors and Initialization (3)

Using the above class definitions, consider the following program

main.cpp
```
#include "foo.hpp"

int main() {
    Derived derived0;
    Derived derived1(123, 456);
}
```

Then the output of this program would be

```
$ ./foo
Base::Base()
Derived::Derived()
Base::Base(int)
Derived::Derived(int, int)
```

Destructors (1)

Similarly to constructors, destructors of derived classes account for the inheritance

1. The body of the destructor is executed
2. The destructors of all non-static members are called in reverse order of declaration
3. The destructors of all direct non-virtual base classes are called in reverse order of construction

The order in which the base class destructors are called is deterministic

- It depends on the order of construction, which in turn only depends on the order of base classes in the *base-specifier-list*

Destructors (2)

Consider the class definitions

foo.hpp
```
struct Base0 {
    ~Base0();
};

struct Base1 {
    ~Base1();
};

struct Derived : Base0, Base1 {
    ~Derived();
};
```

foo.cpp
```
#include "foo.hpp"
#include <iostream>

using namespace std;

Base0::~Base0() {
    cout << "Base0::~Base0()" << endl;
}

Base1::~Base1() {
    cout << "Base1::~Base1()" << endl;
}

Derived::~Derived() {
    cout << "Derived::~Derived()" << endl;
}
```

Destructors (3)

Using the above class definitions, consider the program

main.cpp

```
#include "foo.hpp"

int main() {
    Derived derived;
}
```

Then the output of this program would be

```
$ ./foo
Derived::~Derived()
Base1::~Base1()
Base0::~Base0()
```

Unqualified Name Lookup (1)

It is allowed (although discouraged) to use a name multiple times in an inheritance hierarchy

- Affects unqualified name lookups (lookups without the use of the scope resolution operator `::`)
- A deterministic algorithm decides which alternative matches an unqualified name lookup
- Rule of thumb: Declarations in the derived classes "hide" declarations in the base classes

Multiple inheritance can lead to additional problems even without reusing a name

- In a diamond-shaped inheritance hierarchy, members of the root class appear twice in the most derived class
- Can be solved with *virtual* inheritance
- Should still be avoided whenever possible

Unqualified Name Lookup (2)

Single inheritance example

```
struct A {
    void a();
};

struct B : A {
    void a();
    void b() {
        a();  // calls B::a()
    }
};

struct C : B {
    void c() {
        a();  // calls B::a()
    }
};
```

Unqualified Name Lookup (3)

Diamond inheritance example

```
struct X {
    void x();
};

struct B1 : X { };
struct B2 : X { };

struct D : B1, B2 {
    void d() {
        x();  // ERROR: x is present in B1 and B2
    }
};
```

Qualified Name Lookup

Qualified name lookup can be used to explicitly resolve ambiguities

- Similar to qualified namespace lookups, a class name can appear to the left of the scope resolution operator `::`

```
struct A {
    void a();
};

struct B : A {
    void a();
};

int main() {
    B b;
    b.a();      // calls B::a()
    b.A::a();   // calls A::a()
}
```

Object Representation

The object representation of derived class objects accounts for inheritance

- The base class object is stored as a *subobject* in the derived class object
- Thus, derived classes may still be trivially constructible, copyable, or destructible

foo.cpp

```cpp
struct A {
    int a = 42;
    int b = 123;
};

struct B : A {
    int c = 456;
};

int main() {
    B b;
}
```

foo.o

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $42, -12(%rbp)
    movl    $123, -8(%rbp)
    movl    $456, -4(%rbp)
    movl    $0, %eax
    popq    %rbp
    ret
```

Polymorphic Inheritance

By default, inheritance in C++ is non-polymorphic

- Member definitions in a derived class can hide definitions in the base class
- For example, it matters if we call a function through a pointer to a base object or a pointer to a derived object

```
#include <iostream>

struct Base {
    void foo() { std::cout << "Base::foo()" << std::endl; }
};

struct Derived : Base {
    void foo() { std::cout << "Derived::foo()" << std::endl; }
};

int main() {
    Derived d;
    Base& b = d;

    d.foo();  // prints Derived::foo()
    b.foo();  // prints Base::foo()
}
```

The `virtual` Function Specifier (1)

Used to mark a non-static member function as *virtual*

- Enables dynamic dispatch for this function
- Allows the function to be overriden in derived classes
- A class with at least one virtual function is *polymorphic*

The overridden behavior of the function is preserved even when no compile-time type information is available

- A call to an overridden virtual function through a pointer or reference to a base object will invoke the behavior defined in the derived class
- This behavior is suppressed when qualified name lookup is used for the function call

The virtual Function Specifier (2)

Example

```
#include <iostream>

struct Base {
    virtual void foo() { std::cout << "Base::foo()" << std::endl; }
};
struct Derived : Base {
    void foo() { std::cout << "Derived::foo()" << std::endl; }
};
int main() {
    Base b;
    Derived d;
    Base& br = b;
    Base& dr = d;

    d.foo();         // prints Derived::foo()
    dr.foo();        // prints Derived::foo()
    d.Base::foo();   // prints Base::foo()
    dr.Base::foo();  // prints Base::foo()

    br.foo();        // prints Base::foo()
}
```

Conditions for Overriding Functions (1)

A function overrides a virtual base class function if

- The function name is the same
- The parameter type list (but not the return type) is the same
- The cv-qualifiers of the function are the same
- The ref-qualifiers of the function are the same

If these conditions are met, the function overrides the virtual base class function

- The derived function is also virtual and can be overridden by further-derived classes
- The base class function does not need to be visible
- The return type must be the same or *covariant*

If these conditions are not met, the function may hide the virtual base class function

Conditions for Overriding Functions (2)

Example

```
struct Base {
    private:
    virtual void bar();

    public:
    virtual void foo();
};

struct Derived : Base {
    void bar();           // Overrides Base::bar()
    void foo(int baz);  // Hides Base::foo()
};

int main() {
    Derived d;
    Base& b = d;

    d.foo();  // ERROR: lookup finds only Derived::foo(int)
    b.foo();  // invokes Base::foo();
}
```

The Final Overrider (1)

Every virtual function has a *final overrider*

- The final overrider is executed when a virtual function call is made
- A virtual member function is the final overrider unless a derived class declares a function that overrides it

A derived class can also inherit a function that overrides a virtual base class function through multiple inheritance

- There must only be one final overrider at all times
- Multiple inheritance should be avoided anyway

The Final Overrider (2)

Example

```
struct A {
    virtual void foo();
    virtual void bar();
    virtual void baz();
};
struct B : A {
    void foo();
    void bar();
};
struct C : B {
    void foo();
};
int main() {
    C c;
    A& cr = c;

    cr.foo();  // invokes C::foo()
    cr.bar();  // invokes B::bar()
    cr.baz();  // invokes A::baz()
}
```

The Final Overrider (3)

The final overrider depends on the actual type of an object

```
struct A {
    virtual void foo();
    virtual void bar();
    virtual void baz();
};
struct B : A {
    void foo();
    void bar();
};
struct C : B {
    void foo();
};
int main() {
    B b;
    A& br = b;

    br.foo();  // invokes B::foo()
    br.bar();  // invokes B::bar()
    br.baz();  // invokes A::baz()
}
```

Covariant Return Types (1)

The overriding and base class functions can have *covariant* return types

- Both types must be single-level pointers or references to classes
- The referenced/pointed-to class in the base class function must be a direct or indirect base class of the referenced/pointed-to class in the derived class function
- The return type in the derived class function must be at most as cv-qualified as the return type in the base class function
- Most of the time, the referenced/pointed-to class in the derived class function is the derived class itself

Covariant Return Types (2)

Example

```
struct Base {
    virtual Base* foo();
    virtual Base* bar();
};

struct Derived : Base {
    Derived* foo();  // Overrides Base::foo()
    int bar();       // ERROR: Overrides Base::bar() but has
                     // non-covariant return type
};
```

Construction and Destruction

Virtual functions have to be used carefully during construction and destruction

- During construction and destruction, a class behaves as if no more-derived classes exist
- I.e., virtual function calls during construction and destruction call the final overrider in the constructor's or destructor's class

```
struct Base {
    Base() { foo(); }
    virtual void foo();
};

struct Derived : Base {
    void foo();
};

int main() {
    Derived d;  // On construction, Base::foo() is called
}
```

Virtual Destructors

Derived objects can be deleted through a pointer to the base class

- Undefined behavior unless the destructor in the base class is virtual
- The destructor in a base class should either be protected and non-virtual or public and virtual

```
#include <memory>

struct Base {
    virtual ~Base() { };
};

struct Derived : Base { };

int main() {
    Base* b = new Derived();
    delete b;  // OK
}
```

The override Specifier

The override specifier should be used to prevent bugs

- The override specifier can appear directly after the declarator in a member function declaration or inline member function definition
- Ensures that the member function is virtual and overrides a base class method
- Useful to avoid bugs where a function in a derived class actually hides a base class function instead of overriding it

```
struct Base {
    virtual void foo(int i);
    virtual void bar();
};

struct Derived : Base {
    void foo(float i) override;  // ERROR
    void bar() const override;   // ERROR
};
```

The `final` Specifier (1)

The `final` specifier can be used to prevent overriding a function

- The `final` specifier can appear directly after the declarator in a member function declaration or inline member function definition

```
struct Base {
    virtual void foo() final;
};

struct Derived : Base {
    void foo() override;  // ERROR
};
```

The final Specifier (2)

The final specifier can be used to prevent inheritance from a class

- The final specifier can appear in a class definition, immediately after the class name

```
struct Base final {
    virtual void foo();
};

struct Derived : Base {  // ERROR
    void foo() override;
};
```

Abstract Classes (1)

C++ allows abstract classes which cannot be instantiated, but used as a base class

- Any class which declares or inherits at least one *pure virtual* function is an abstract class
- A pure virtual member function declaration contains the sequence `= 0` after the declarator and `override`/`final` specifiers
- Pointers and references to an abstract class can be declared

A definition can still be provided for a pure virtual function

- Derived classes can call this function using qualified name lookup
- The pure specifier `= 0` cannot appear in a member function definition (i.e. the definition can not be provided inline)

Making a virtual function call to a pure virtual function in the constructor or destructor of an abstract class is **undefined behavior**

Abstract Classes (2)

Example

```
struct Base {
    virtual void foo() = 0;
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Base b;          // ERROR
    Derived d;
    Base& dr = d;
    dr.foo();        // calls Derived::foo()
}
```

Abstract Classes (3)

A definition may be provided for a pure virtual function

```
struct Base {
    virtual void foo() = 0;
};

void Base::foo() { /* do something */ }

struct Derived : Base {
    void foo() override { Base::foo(); }
};
```

Abstract Classes (4)

The destructor may also be marked as pure virtual

- Useful when a class needs to be abstract, but has no suitable functions that could be declared pure virtual
- In this case a definition *must* be provided

```
struct Base {
    virtual ~Base() = 0;
};

Base::~Base() { }

int main() {
    Base b;  // ERROR
}
```

Abstract Classes (5)

Abstract classes cannot be instantiated

- Programs have to refer to abstract classes through pointers or references
- Smart pointers (owning), references (non-owning), or raw pointers (if `nullptr` is possible)

```
#include <memory>

struct Base {
    virtual ~Base();
    virtual void foo() = 0;
};

struct Derived : Base { void foo() override; };

void bar(const Base& b) { b.foo(); }

int main() {
    std::unique_ptr<Base> b = std::make_unique<Derived>();
    b->foo();  // calls Derived::foo()

    bar(*b);   // calls Derived::foo() within bar
}  // destroys b, undefined behavior unless ~Base() is virtual
```

dynamic_cast (1)

Converts pointers and references to classes in an inheritance hierarchy

- Syntax: `dynamic_cast < new_type > ( expression )`
- *new_type* may be a pointer or reference to a class type
- *expression* must be an lvalue expression of reference type if *new_type* is a reference type, and an rvalue expression of pointer type otherwise

Most common use case: Safe downcasts in an inheritance hierarchy

- Involves a runtime check whether *new_type* is a base of the actual polymorphic type of *expression*
- If the check fails, returns `nullptr` for pointer types, and throws an exception for reference types
- Requires runtime type information which incurs some overhead

For other use cases: See the reference documentation

dynamic_cast (2)

Example

```
struct A {
    virtual ~A() = default;
};

struct B : A {
    void foo() const;
};

struct C : A {
    void bar() const;
};

void baz(const A* aptr) {
    if (const B* bptr = dynamic_cast<const B*>(aptr)) {
        bptr->foo();
    } else if (const C* cptr = dynamic_cast<const C*>(aptr)) {
        cptr->bar();
    }
}
```

dynamic_cast (3)

dynamic_cast has a non-trivial performance overhead

- Notable impact if many casts have to be performed
- Alternative: Use a type enum in conjunction with static_cast

```
struct Base {
    enum class Type {
        Base,
        Derived
    };

    Type type;

    Base() : type(Type::Base) { }
    Base(Type type) : type(type) { }

    virtual ~Base();
};

struct Derived : Base {
    Derived() : Base(Type::Derived) { }
};
```

dynamic_cast (4)

Example (continued)

```
void bar(const Base* basePtr) {
    switch (basePtr->type) {
    case Base::Type::Base:
        /* do something with Base */
        break;
    case Base::Type::Derived:
        const Derived* derivedPtr
            = static_cast<const Derived*>(basePtr);

        /* do something with Derived */

        break;
    }
}
```

Vtables (1)

Polymorphism does not come for free

- Dynamic dispatch has to be implemented somehow
- The C++ standard does not prescribe a specific implementation
- Compilers typically use *vtables* to resolve virtual function calls

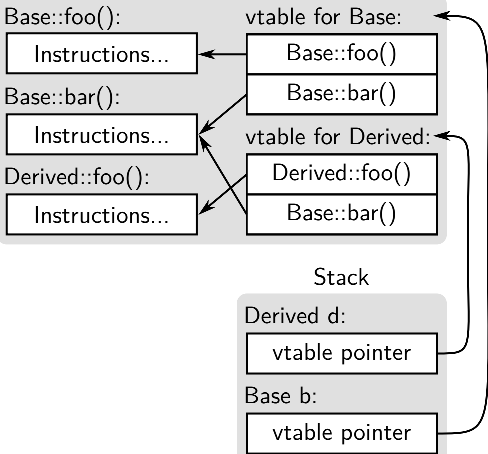
Vtables setup and use

- One vtable is constructed per class with virtual functions
- The vtable contains the addresses of the virtual functions of that class
- Objects of classes with virtual functions contain an additional pointer to the base of the vtable
- When a virtual function is invoked, the pointer to the vtable is followed and the function that should be executed is resolved

Vtables (2)

Example

```
struct Base {
    virtual void foo();
    virtual void bar();
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Base b;
    Derived d;

    Base& br = b;
    Base& dr = d;

    br.foo();
    dr.foo();
}
```



Performance Implications

Virtual function calls incur an additional indirection

- The pointer to the vtable is followed
- The pointer to the actual function is followed
- Each step may incur a cache miss
- Can be very notable when invoking a virtual function millions of times

Polymorphic objects have larger size

- Each object of a polymorphic class needs to store a pointer to the vtable
- In our example, both `Base` and `Derived` occupy 8 bytes of memory despite having no data members

Inheritance Modes

Recall the definition of a *base-specifier*

access-specifier virtual-specifier base-class-name

The *access-specifier* specifies the *inheritance mode*

- The inheritance mode controls the access mode of base class members in the derived class
- If no *access-specifier* is given, derived classes defined with `struct` have `public` inheritance mode by default
- If no *access-specifier* is given, derived classes defined with `class` have `private` inheritance mode by default

Public Inheritance (1)

Semantics

- Public base class members are usable as public members of the derived class
- Protected base class members are usable as protected members of the derived class

Models the subtyping (IS-A) relationship of object-oriented programming

- Pointers and references to a derived object should be usable wherever a pointer to the a base object is expected
- A derived class must maintain the class invariants of its base classes
- A derived class must not strengthen the preconditions of any member function it overrides
- A derived class must not weaken the postconditions of any member function it overrides

Public Inheritance (2)

Example

```
class A {
    protected:
    int a;

    public:
    int b;
};

class B : public A {
    public:
    void foo() {
        return a + 42;  // OK: a is usable as protected member of B
    }
};

int main() {
    B b;
    b.b = 42;  // OK: b is usable as public member of B
    b.a = 42;  // ERROR: a is not visible
}
```

Private Inheritance (1)

Semantics

- Public base class members are usable as private members of the derived class
- Protected base class members are usable as private members of the derived class

Some specialized use cases

- Policy-based design using templates (more details later)
- Mixins
- Model composition if some requirements are met
  - The base object needs to be constructed or destructed before or after some object in the derived object
  - The derived class needs access to protected members of the base class
  - The derived class needs to override virtual methods in the base class

Private Inheritance (2)

Example

```
class A {
    protected:
    A(int);  // Constructor is protected for some reason
};

class C : private A {
    public:
    C() : A(42) { }

    const A& getA() {  // Act as if we have a member of type A
        return *this;
    }
};
```

Protected Inheritance (1)

Semantics

- Public base class members are usable as protected members of the derived class
- Protected base class members are usable as protected members of the derived class
- Within the derived class and all further-derived classes, pointers and references to a derived object may be used where a pointer or reference to the base object is expected

Models “controlled polymorphism”

- Mainly used for the same purposes as private inheritance, where inheritance should be shared with subclasses
- Rarely seen in practice

Protected Inheritance (2)

Example

```
class A {
    protected:
    int a;

    public:
    int b;
};

class B : protected A {
    public:
    void foo() {
        return a + 42;  // OK: a is usable as protected member of B
    }
};

int main() {
    B b;
    b.b = 42;  // ERROR: b is not visible
    b.a = 42;  // ERROR: a is not visible
}
```

Multiple Inheritance

C++ supports multiple inheritance

- Rarely required
- Easy to produce convoluted code
- Leads to implementation issues (e.g. diamond-inheritance)

There are C++ language features to address such issues

- You will likely never need multiple inheritance during this lecture
- For details: Check the reference documentation
- Multiple inheritance should be avoided whenever possible

Exceptions in C++

C++ supports exceptions with similar semantics as other languages

- Exceptions transfer control and information up the call stack
- Can be thrown by `throw`-expressions, `dynamic_cast`, `new`-expressions and some standard library functions

While transferring control up the call stack, C++ performs *stack unwinding*

- Properly cleans up all objects with automatic storage duration
- Ensures correct behavior e.g. of RAII classes

Exceptions do not have to be handled

- Can be handled in `try`-`catch` blocks
- Unhandled exceptions lead to termination of the program though
- Errors during exception handling also lead to termination of the program

Throwing Exceptions

Objects of any complete type may be thrown as exception objects

- Usually exception objects should derive directly or indirectly from `std::exception`, and contain information about the error condition
- Syntax: `throw` *`expression`*
- Copy-initializes the exception object from *`expression`* and throws it

```
#include <exception>

void foo(unsigned i) {
    if (i == 42)
        throw 42;

    throw std::exception();
}
```

Handling Exceptions

Exceptions are handled in try-catch blocks

- Exceptions that occur while executing the try-block can be handled in the catch-blocks
- The parameter type of the catch-block determines which type of exception causes the block to be entered

```
#include <exception>

void bar() {
    try {
        foo(42);
    } catch (int i) {
        /* handle exception */
    } catch (const std::exception& e) {
        /* handle exception */
    }
}
```

Usage Guidelines

Exceptions should only be used in rare cases

- Main legitimate use case: Failure to (re)establish a class invariant (e.g. failure to acquire a resource in an RAII constructor)
- Functions should **not** throw exceptions when preconditions are not met – use assertions instead
- Exceptions should **not** be used for control flow

Some functions must not throw exceptions

- Destructors
- Move constructors and assignment operators
- See reference documentation for details

Generally, exceptions should be avoided where possible

Templates

Motivation

Functionality is often independent of a specific type `T`

- E.g. `swap(T& a, T& b)`
- E.g. `std::vector<T>`
- Many more examples (e.g. in exercises)

Functionality should be available for all suitable types `T`

- How to avoid massive code duplication?
- How to account for user-defined types?

Templates

A *template* defines a family of classes, functions, type aliases, or variables

- Templates are parameterized by one or more template parameters
  - Type template parameters
  - Non-type template parameters
  - Template template parameters
- In order to use a template, *template arguments* need to be provided
  - Template arguments are substituted for the template parameters
  - Results in a *specialization* of the template
- Templates are a *compile-time* construct
  - When used (inaccurate, more details soon), templates are *instantiated*
  - Template instantiation actually compiles the code for the respective specialization

Example

(Simplified) definition of `std::vector`

```
class A;
//----------------------------------------------------------------
template <class T>  // T is a type template parameter
class vector {
    public:
    /* ... */
    void push_back(const T& element);
    /* ... */
};
//----------------------------------------------------------------
int main() {
    vector<int> vectorOfInt; // int is substituted for T
    vector<A> vectorOfA;     // A is substituted for T
}
```

Template Syntax

Several C++ entities can be declared as templates

- Syntax: `template < parameter-list > declaration`

parameter-list is a comma-separated list of template parameters

- Type template parameters
- Non-type template parameters
- Template template parameters

declaration is one of the following declarations

- `class`, `struct` or `union`
- A nested member class or enumeration type
- A function or member function
- A static data member at namespace scope or a data member at class scope
- A type alias

Type Template Parameters

Type template parameters are placeholders for arbitrary types

- Syntax: `typename` *name* or `class` *name*
- *name* may be omitted (e.g. in forward declarations)
- There is no difference between using `typename` or `class`
- In the body of the template declaration, *name* is a type alias for the type supplied during instantiation

```
template <class, class>
struct Baz;
//----------------------------------------------------------------
template <class T>
struct Foo {
    T bar(T t) {
        return t + 42;
    }
};
```

Non-Type Template Parameters

Non-type template parameters are placeholders for certain values

- Syntax: *type name*
- *name* may be omitted (e.g. in forward declarations)
- *type* may be an integral type, pointer type, enumeration type or lvalue reference type
- Within the template body, *name* of a non-type parameter can be used in expressions

```
template <class T, size_t N>
class Array {
    T storage[N];

    public:
    T& operator[](size_t i) {
        assert(i < N);
        return storage[i];
    }
};
```

Template Template Parameters

Type template parameters can themselves be templated

- Syntax: `template` < *parameter-list* > `typename` *name* or `template` < *parameter-list* > `class` *name*
- *name* may be omitted (e.g. in forward declarations)
- Within the template body, *name* is a template name, i.e. it needs template arguments to be instantiated

```
template <template <class, size_t> class ArrayType>
class Foo {
    ArrayType<int, 42> someArray;
};
```

Rarely used or required, should be avoided whenever possible

Default Template Arguments

All three types of template parameters can have default values

- Syntax: *template-parameter = default*
- *default* must be a type name for type and template template parameters, and a literal for non-type template parameters
- Template parameters with default values may not be followed by template parameters without default values

```
template <typename T = std::byte, size_t Capacity = 1024>
class Buffer {
    T storage[Capacity];
};
```

Using Templates

In order to use a templated entity, template arguments need to be provided

- Syntax: *template-name* < *parameter-list* >
- *template-name* must be an identifier that names a template
- *parameter-list* is a comma-separated list of template arguments
- Results in a *specialization* of the template

Template arguments must match the template parameters

- At most as many arguments as parameters
- One argument for each parameter without a default value

In some cases, template arguments can be *deduced* automatically.

Type Template Arguments

Template arguments for type template parameters must name a type (which may be incomplete)

```
class A;
//------------------------------------------------------------
template <class T1, class T2 = int, class T3 = double>
class Foo { };
//------------------------------------------------------------
int main() {
    Foo<int> foo1;
    Foo<A> foo2;
    Foo<A*> foo3;
    Foo<int, A> foo4;
    Foo<int, A, A> foo5;
}
```

Non-Type Template Arguments (1)

Template arguments for non-type template parameters must be (converted) constant expressions

- Converted constant expressions can be evaluated at compile-time
- May incur a limited set of implicit conversions
- The (possibly implicitly converted) type of the expression must match the type of the template parameter

Restrictions for non-type template parameters of reference or pointer type

- May not refer to a subobject (non-static class member, base subobject)
- May not refer to a temporary object
- May not refer to a string literal

Non-Type Template Arguments (2)

Example

```
//------------------------------------------------------------------
template <unsigned N>
class Foo { };
//------------------------------------------------------------------
int main() {
    Foo<42u> foo1; // OK: no conversion
    Foo<42> foo2;  // OK: numeric conversion
}
```

constexpr

Functions or variables cannot be evaluated at compile time by default

- Use the constexpr keyword to indicate that the value of a function or variable can be evaluated at compile time
- constexpr variables must have literal type and be immediately initialized
- constexpr functions must have literal return and parameter types

```
#include <array>
//---------------------------------------------------------------
class Element { /* ... */ };
//---------------------------------------------------------------
class Foo {
    static constexpr size_t numElements = 42;
    constexpr size_t calculateBufferSize(size_t elements) {
        return elements * sizeof(Element);
    }

    std::array<std::byte, calculateBufferSize(numElements)> array;
};
```

Template Template Arguments

Arguments to template template arguments must name a class template or template alias

```
#include <array>
//------------------------------------------------------------------
template <class T, size_t N>
class MyArray { };
//------------------------------------------------------------------
template <template<class, size_t> class Array>
class Foo {
    Array<int, 42> bar;
};
//------------------------------------------------------------------
int main() {
    Foo<MyArray> foo1;
    Foo<std::array> foo2;
}
```

Example: Class Templates

```
template <class T, size_t N>
class MyArray {
    private:
    T storage[N];

    public:
    /* ... */

    T& operator[](size_t index) {
        return storage[index];
    }

    const T& operator[](size_t index) const {
        return storage[index];
    }

    /* ... */
};
```

Example: Function Templates

```
class A { };
//------------------------------------------------------------------
template <class T>
void swap(T& a, T& b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
//------------------------------------------------------------------
int main() {
    A a1;
    A a2;

    swap<A>(a1, a2);
    swap(a1, a2);    // Also OK: Template arguments are deduced
}
```

Example: Alias Templates

```
namespace something::extremely::nested {
//----------------------------------------------------------------
template <class T, class R>
class Handle { };
//----------------------------------------------------------------
}  // namespace something::extremely::nested
//----------------------------------------------------------------
template <typename T>
using Handle = something::extremely::nested::Handle<T, void*>;
//----------------------------------------------------------------
int main() {
    Handle<int> handle1;
    Handle<double> handle2;
}
```

Example: Variable Templates

```
template <class T>
constexpr T pi = T(3.1415926535897932385L);
//------------------------------------------------------------------
template <class T>
T area(T radius) {
    return pi<T> * radius * radius;
}
//------------------------------------------------------------------
int main() {
    double a = area<double>(1.0);
}
```

Example: Class Member Templates

```
#include <iostream>
#include <array>
//--------------------------------------------------------------
struct Foo {
    template <class T>
    using ArrayType = std::array<T, 42>;

    template <class T>
    void printSize() {
        std::cout << sizeof(T) << std::endl;
    }
};
//--------------------------------------------------------------
int main() {
    Foo::ArrayType<int> intArray;

    Foo foo;
    foo.printSize<Foo::ArrayType<int>>();
}
```

Template Instantiation

A function or class template by itself is not a type, an object, or any other entity

- No assembly is generated from a file that contains only template definitions
- A template specialization must be *instantiated* for any assembly to appear

Template instantiation

- Compiler generates an actual function or class for a template specialization
- Explicit instantiation: Explicitly request instantiation of a specific specialization
- Implicit instantiation: Use a template specialization in a context that requires a complete type

Explicit Template Instantiation (1)

Forces instantiation of a template specialization

- Class template syntax
  - `template class` *template-name* `<` *argument-list* `>;`
  - `template struct` *template-name* `<` *argument-list* `>;`
- Function template syntax
  - `template` *return-type name* `<` *argument-list* `>` `(`*parameter-list*`);`

Explanation

- Explicit instantiations have to follow the one definition rule
- Generates assembly for the function specialization or class specialization and all its member functions
- Template definition must be visible at the point of explicit instantiation
- Template definition and explicit instantiation should usually be placed in implementation file

Explicit Template Instantiation (2)

Example

```
template <class T>
struct A {
    T foo(T value) { return value + 42; }

    T bar() { return 42; }
};
//------------------------------------------------------------------
template <class T>
T baz(T a, T b) {
    return a * b;
}
//------------------------------------------------------------------
// Explicit instantiation of A<int>
template struct A<int>;
// Explicit instantiation of baz<float>
template float baz<float>(float, float);
```

Implicit Template Instantiation (1)

Using a template specialization in a context that requires a complete type triggers implicit instantiation

- Only if the specialization has not been explicitly instantiated
- Members of a class template are only implicitly instantiated if they are actually used

The definition of a template must be visible at the point of implicit instantiation

- Definitions must usually be provided in the *header* file if implicit instantiation is desired

Implicit Template Instantiation (2)

Example

```
template <class T>
struct A {
    T foo(T value) {
        return value + 42;
    }

    T bar();
};
//--------------------------------------------------------------
int main() {
    A<int> a;              // Instantiates only A<int>
    int x = a.foo(32);     // Instantiates A<int>::foo

    // No error although A::bar is never defined

    A<float>* aptr;        // Does not instantiate A<float>
}
```

Differences between Explicit and Implicit Instantiation

Implicit instantiation

- Pro: Template can be used with any suitable type
- Pro: No unnecessary assembly is generated
- Con: Definition has to be provided in header
- Con: *User* of our templates has to compile them

Explicit instantiation

- Pro: Explicit instantiations can be compiled into library
- Pro: Definition can be encapsulated in source file
- Con: Limits usability of our templates

Usually, we do not need to explicitly instantiate our templates

Instantiation Caveats

The compiler actually generates code for instantiations

- Code is generated for *each* instantiation with different template arguments
- Conceptually, template parameters are replaced by template arguments
- If one instantiation generates 1 000 lines of assembly, 10 instantiations generate 10 000 lines of assembly
- Can substantially increase compilation time

Instantiations are generated locally for each compilation unit

- Templates are implicitly `inline`
- The same instantiation can exist in different compilation units without violating ODR

Inline vs. Out-of-Line Definition

Out-of-line definitions should be preferred even when defining class templates in headers

- Improves readability of interface
- Requires somewhat “weird” syntax

```
template <class T>
struct A {
    T value;

    A(T value);

    template <class R>
    R convert();
};
//--------------------------------------------------------------
template <class T>
A<T>::A(T value) : value(value) { }
//--------------------------------------------------------------
template <class T>
template <class R>
R A<T>::convert() { return static_cast<R>(value); }
```

Concepts and Constraints (1)

A programmer can try to instantiate a template with any arguments

- A template might assume certain things about its parameters (e.g. presence of a member function)
- Without further programmer intervention, these assumptions are *implicit*
- In this case, the behavior of templates resembles duck typing
- May lead to (horrible) compile-time errors when used with incorrect template arguments

```
template <class T>
struct A {
    int bar(T t) { return t.foo(); }
};
//--------------------------------------------------------------
int main() {
    A<int> b;   // OK: A<int>::bar is not instantiated
    b.bar(42);  // ERROR: int does not have foo member
}
```

Concepts and Constraints (2)

Constraints and concepts *explicitly* specify requirements on template parameters

- Allows the compiler to check requirements
- Allows the compiler to generate much more informative error messages
- Greatly improves safety (*explicit* concepts instead of *implicit* assumptions)
- Can still only check syntactic requirements

The standard library provides a large set of useful concepts

- Defined in the `<concepts>` and other standard library headers
- Specify syntactic requirements on a number of types (e.g. `std::swappable_with`)
- Most of the time also specify semantic requirements
- Concepts can also be user-defined (beyond the scope of this lecture)

Concepts and Constraints (3)

Constraints can be applied to template arguments or function declarations

```
#include <concepts>

template <typename T> requires std::floating_point<T>
T fdiv1(T a, T b) {
    return a / b;
}

template <typename T>
T fdiv2(T a, T b) requires std::floating_point<T> {
    return a / b;
}

template <std::floating_point T>
T fdiv3(T a, T b) {
    return a / b;
}
```

Concepts and Constraints (4)

Constraints are extremely useful to make working with templates somewhat easier

- Unfortunately not used much in the standard library
- Should be used as much as possible in your code
- Any assumptions about template parameters should be made explicit

We only scratched the surface of concepts and constraints

- Many more details in the reference documentation
- We will return to concepts and constraints towards the end of this course
- Meanwhile: Learn to read the error messages produced by constraints
- E.g. through the compiler explorer at https://compiler.db.in.tum.de

Dependent Names (1)

Within a class template, some names may be deduced to refer to the current instantiation

- The class name itself (without template parameters)
- The name of a member of the class template
- The name of a nested class of the class template

```
template <class T>
struct A {
    struct B { };

    B* b;  // B refers to A<T>::B

    A(const A& other);  // A refers to A<T>

    void foo();
    void bar() {
        foo();  // foo refers to A<T>::foo
    }
};
```

Dependent Names (2)

Names that are members of templates are not considered to be types by default

- When using a name that is a member of a template outside of any template declaration or definition
- When using a name that is not a member of the current instantiation within a template declaration or definition
- If such a name should be considered as a type, the `typename` disambiguator has to be used

In some context, only type names can validly appear

- Allows the `typename` disambiguator to be omitted
- A detailed list of such cases can be found in the reference documentation

Dependent Names (3)

Example

```
struct A {
    using MemberTypeAlias = float;
};
//------------------------------------------------------------
template <class T>
struct B {
    // no disambiguator required
    using AnotherMemberTypeAlias = T::MemberTypeAlias;

    // disambiguator required
    typename T::MemberTypeAlias* ptr;
};
//------------------------------------------------------------
int main() {
    // value has type float
    B<A>::AnotherMemberTypeAlias value = 42.0f;
}
```

Dependent Names (4)

Similar rules apply to template names within template definitions

- Any name that is not a member of the current instantiation is not considered to be a template name
- If such a name should be considered as a template name, the `template` disambiguator has to be used

```
template <class T>
struct A {
    template <class R>
    R convert(T value) { return static_cast<R>(value); }
};
//------------------------------------------------------------
template <class T>
T foo() {
    A<int> a;

    return a.template convert<T>(42);
}
```

Reference Collapsing

Templates and type aliases may form references to references

```
template <class T>
class Foo {
    using Trref = T&&;
};

int main() {
    Foo<int&&>::Trref x; // what is the type of x?
}
```

Reference collapsing rules apply

- Rvalue reference to rvalue reference collapses to rvalue reference
- Any other combination forms an lvalue reference

Explicit Template Specialization

We may want to modify the behavior of templates for specific template arguments

- For example, a templated `find` method can employ different algorithms on arrays (binary search) vs. linked lists (linear search)

We can explicitly specialize templates to achieve this

- Define specific implementations for certain template arguments
- All template arguments can be specified (full specialization)
- Some template arguments can be specified (partial specialization)

Full Specialization

Defines a specific implementation for a full set of template arguments

- Has to appear after the declaration of the original template
- Syntax: `template <>` *declaration*
- Most types of templates can be fully specialized

```
template <class T>
class MyContainer {
    /* generic implementation */
};
//-----------------------------------------------------------------
template <>
class MyContainer<long> {
    /* specific implementation */
};
//-----------------------------------------------------------------
int main() {
    MyContainer<float> a;  // uses generic implementation
    MyContainer<long> b;   // uses specific implementation
}
```

Partial Specialization

Defines a specific implementation for a partial set of template arguments

- Has to appear after the declaration of the original template
- `template` < *parameter-list* > `class` *name* < *argument-list* >
- `template` < *parameter-list* > `struct` *name* < *argument-list* >
- Only class templates can be partially specialized
- Function overloads can simulate function template specialization

```
template <class C, class T>
class SearchAlgorithm {
    void find (const C& container, const T& value) {
        /* do linear search */
    }
};
//-------------------------------------------------------------
template <class T>
class SearchAlgorithm<std::vector<T>, T> {
    void find (const std::vector<T>& container, const T& value) {
        /* do binary search */
    }
};
```

Template Argument Deduction

Some template arguments for class and function templates can be *deduced*

- All template arguments have to be known to instantiate a class or function template
- Not all template arguments have to be specified for class and function templates
- Template arguments can be omitted entirely quite frequently
- Makes it possible, for example, to use template operators

```
template <class T>
void swap(T& a, T& b);
//--------------------------------------------------------------
int main() {
    int a = 0;
    int b = 42;

    swap(a, b);  // T is deduced to be int
}
```

Function Template Argument Deduction

Deduces template arguments in function calls

- Attempts to deduce the template arguments based on the types of the function arguments
- Argument deduction may fail if ambiguous types are deduced
- Highly complex set of rules (see reference documentation)

```
template <class T>
T max(const T& a, const T& b);
//------------------------------------------------------------
int main() {
    int a = 0;
    long b = 42;

    max(a, b);          // ERROR: Ambiguous deduction of T
    max(a, a);          // OK
    max<int>(a, b);     // OK
    max<long>(a, b);    // OK
}
```

Class Template Argument Deduction

Deduces class template arguments in some cases

- Declarations that also specify initialization of a variable
- new-expressions
- Attempts to deduce the template arguments based on the types of the constructor arguments

```
#include <memory>
//--------------------------------------------------------------
template <class T>
struct Foo {
    Foo(T t);
};
//--------------------------------------------------------------
int main() {
    Foo foo(12);
    std::unique_ptr ptr = make_unique<int>(42);
}
```

The auto Type (1)

The auto placeholder can be used to deduce the type of a variable from its initializer

- Deduction follows the same rules as function template argument deduction
- auto may be accompanied by the usual modifiers such as const, * or &
- Extremely convenient when using complex types (such as standard library iterators)

```
#include <unordered_map>
//-------------------------------------------------------------
int main() {
    std::unordered_map<int, const char*> intToStringMap;

    std::unordered_map<int, const char*>::iterator it1 =
        intToStringMap.begin();  // noone wants to read this

    auto it2 = intToStringMap.begin();  // much better
}
```

The auto Type (2)

auto does not require any modifiers to work

- Can make code more error prone and hard to understand
- All known modifiers should always be added to auto

```
const int** foo();
//--------------------------------------------------------------
int main() {
    // BAD:
    auto f1 = foo();         // auto is const int**
    const auto f2 = foo();   // auto is int**
                             // f2 has type int const** const
    auto** f3 = foo();       // auto is const int

    // GOOD:
    const auto** f4 = foo();  // auto is int
}
```

The auto Type (3)

auto is not deduced to a reference type

- Might incur unwanted copies
- All known modifiers should always be added to auto

```
struct A {
    const A& foo() { return *this; }
};
//--------------------------------------------------------------
int main() {
    A a;
    auto a1 = a.foo();          // BAD:  auto is const A, copy
    const auto& a2 = a.foo()   // GOOD: auto is A, no copy
}
```

Structured Bindings (1)

Binds some names to subobjects or elements of the initializer

- Syntax (1): `auto [ identifier-list ] = expression;`
- Syntax (2): `auto [ identifier-list ]( expression );`
- Syntax (3): `auto [ identifier-list ]{ expression };`
- `auto` may be cv- or reference-qualified

Explanation

- The identifiers in *identifier-list* are bound to the subobjects or elements of the initializer
- Can bind to arrays, tuple-like types and accessible data members
- Very useful during iteration, especially over associative containers

Structured Bindings (2)

Example

```
#include <utility>
//------------------------------------------------------------
struct Foo {
    float y;
    long z;
};
//------------------------------------------------------------
std::pair<int, long> bar();
//------------------------------------------------------------
int main() {
    Foo foo;
    int array[4];

    auto [a1, a2, a3, a4] = array;  // copies array, a1 - a4 refer to copy
    auto& [y, z] = foo;   // y refers to foo.y, z refers to foo.z
    auto [l, r] = bar();  // move-constructs pair p, l refers to p.first,
                          // r refers to p.second
}
```

Parameter Packs (1)

Parameter packs are template parameters that accept zero or more arguments

- Non-type: *type* ... *Args*
- Type: `typename|class ...` *Args*
- Template:
  `template <` *parameter-list* `> typename|class ...` *Args*
- Can appear in alias, class and function template parameter lists
- Templates with at least on parameter pack are called *variadic templates*

Function parameter packs

- Appears in the function parameter list of a variadic function template
- Syntax: *Args* ... *args*

Parameter pack expansion

- Syntax: *pattern* ...
- Expands to a comma-separated list of *pattern*s (*pattern* must contain at least one parameter pack)

Parameter Packs (2)

```
template <typename... T>
struct Tuple { };
//-------------------------------------------------------------
template <typename... T>
void printTuple(const Tuple<T...>& tuple);
//-------------------------------------------------------------
template <typename... T>
void printElements(const T&... args);
//-------------------------------------------------------------
int main() {
    Tuple<int, int, float> tuple;

    printTuple(tuple);
    printElements(1, 2, 3, 4);
}
```

Parameter Packs (3)

Implementation of variadic templates is somewhat involved

- Most straightforward way: Tail recursion (usually optimized away)

```
#include <iostream>
//------------------------------------------------------------
void printElements() { }
//------------------------------------------------------------
template <typename Head, typename... Tail>
void printElements(const Head& head, const Tail&... tail) {
    std::cout << head;

    if constexpr (sizeof...(tail) > 0)
        std::cout << ", ";

    printElements(tail...);
}
//------------------------------------------------------------
int main() {
    printElements(1, 2, 3.0, 3.14, 4);
}
```

Fold Expressions (1)

Reduces a parameter pack over a binary operator *op*

- Syntax (1): `(` *pack op* `... )`
- Syntax (2): `( ...` *op pack* `)`
- Syntax (3): `(` *pack op* `...` *op init* `)`
- Syntax (4): `(` *init op* `...` *op pack* `)`
- *pack* must be an expression that contains an unexpanded parameter pack
- *init* must be an expression that does not contain a parameter pack

Semantics

- $(E \circ \ldots)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ E_n))$
- $(\ldots \circ E)$ becomes $((E_1 \circ E_2) \circ \ldots) \circ E_n$
- $(E \circ \ldots \circ I)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ (E_n \circ I)))$
- $(I \circ \ldots \circ E)$ becomes $(((I \circ E_1) \circ E_2) \circ \ldots) \circ E_n$

Fold Expressions (2)

Enables more concise implementation of variadic templates in some cases

```
template <typename R, typename... Args>
R reduceSum(const Args&... args) {
    return (args + ...);
}
//------------------------------------------------------------
int main() {
    return reduceSum<int>(1, 2, 3, 4);  // returns 10
}
```

Concise implementations quickly become concise but extremely hard to understand

- Only used in some specialized cases

Template Metaprogramming

Templates are instantiated at *compile-time*

- Allows "programming" at compile-time (*template metaprogramming*)
- Templates are actually a Turing-complete (sub-)language
- Allows for very useful but at times very involved tricks (e.g. type traits)

```
template <unsigned N>
struct Factorial {
    static constexpr unsigned value = N * Factorial<N - 1>::value;
};
//-----------------------------------------------------------
template <>
struct Factorial<0> {
    static constexpr unsigned value = 1;
};
//-----------------------------------------------------------
int main() {
    return Factorial<6>::value;  // computes 6! at compile time
}
```

static_assert

The `static_assert` declaration checks assertions at compile-time

- Syntax (1): `static_assert` (*bool-constexpr*)
- Syntax (2): `static_assert` (*bool-constexpr*, *message*)
- *bool-constexpr* must be a constant expression that evaluates to `bool`
- *message* may be a string that appears as a compiler error if *bool-constexpr* is `false`

```
template <unsigned N>
class NonEmptyBuffer {
    static_assert(N > 0);
};
```

Type Traits

Type traits compute information about types at compile time

- Simple form of template metaprogramming
- E.g. `std::numeric_limits` is a type trait

```
template <typename T>
struct IsUnsigned {
    static constexpr bool value = false;
};
//----------------------------------------------------------
template <>
struct IsUnsigned <unsigned char> {
    static constexpr bool value = true;
};
/* Further specializations of IsUnsigned for all unsigned types */
//----------------------------------------------------------
template <typename T>
void foo() {
    static_assert(IsUnsigned<T>::value);
}
```

C++ provides many useful type traits (see reference documentation)

Standard Library I

The Standard Library

Provides a collection of useful C++ classes and functions

- Is itself implemented in C++
- Part of the ISO C++ standard
  - Defines interface, semantics and contracts the implementation has to abide by (e.g. runtime complexity)
  - Implementation is *not* part of the standard
  - Multiple vendors provide their own implementations
  - Best known: `libstdc++` (used by gcc) and `libc++` (used by llvm)
- All features are declared within the `std` namespace
- Functionality is divided into sub-libraries each consisting of multiple headers
- Includes parts of the C standard library
  - For backward compatibility
  - Headers begin with "c" (e.g. `cstring`)
  - C++ standard library functions should always be preferred

The Standard Library - Feature Overview (1)

Most important library features:

- Utilities
  - Memory management (`new`, `delete`, `unique_ptr`, `shared_ptr`)
  - Error handling (exceptions, `assert()`)
  - Time (clocks, durations, timestamps, ...)
  - Optionals, Variants, Tuples, ...
- Strings
  - String class
  - String views
  - C-style string handling
- Containers: `array`, `vector`, lists, maps, sets
- Algorithms: (stable) sort, search, max, min, ...
- Iterators
- Numerics
  - Common mathematical functions (`sqrt`, `pow`, `mod`, `log`, ...)
  - Complex numbers
  - Random number generation

The Standard Library - Feature Overview (2)

- I/O
  - Input-/Output streams
  - File streams
  - String streams
- Threads
  - Thread class
  - (shared) mutexes
  - futures
- And much more
  - Localization
  - Regex
  - Atomics
  - Filesystem support
  - ...

std::string

std::string is a class encapsulating character sequences

- Manages its own memory (so no need for new/malloc/unique_ptr)
- Has a wide array of member functions, making string manipulation easier
- Knows its own length: No need to worry about null termination!
- Contents are guaranteed to be stored in memory contiguously
- Can be used like a C-style char pointer
- Access to the underlying C-style char pointer via c_str()

std::string is defined in the <string> library header

- It is a type alias to std::basic_string<char>
- std::basic_string also has specializations for 16- and 32-bit character strings
- Specialization of std::basic_string with custom character types possible

std::string should *always* be preferred over char pointers!

Creating a std::string

The default constructor creates an empty string of length 0

```
std::string s;
s.size(); // == 0
```

Creation from a string literal via constructor argument or assignment

```
std::string s_constructed("my literal");
std::string s_assigned = "my literal";
```

Take care with strings that contain null-bytes:

```
std::string s = "null\0byte!";
std::cout << s << std::endl; // prints "null"

std::string s_with_size("null\0byte!", 10);
std::cout << s_with_size << std::endl; // prints "nullbyte!"
```

Accessing contents of `std::string` (1)

Single characters can be accessed with the subscript operator

```
std::string s = "Hello World!";
std::cout << s[4] << s[6] << std::endl; // prints "oW"
```

Since it returns a reference, this can be used to modify the string

```
std::string s = "Hello World!";
s[4] = 'x';
s[6] = 'Y';
s[10] = s[9];
std::cout << s << std::endl; // prints "Hellx Yorll!"
```

Out of bounds access with array notation results in undefined behavior

Accessing contents of std::string (2)

Iterators allow iteration over contents

```
std::string s = "Hello World!";
for (auto iter = s.begin(); iter != s.end(); ++iter) {
    ++(*iter);
}
std::cout << s << std::endl; // prints "Ifmmp!Xpsme"
```

For backwards compatibility: `c_str()` returns null-terminated char pointer

```
int i_only_know_c(const char* str) {
    int len = 0;
    while (str) { str++; len++; }
    return len;
}

std::string i_am_modern_cpp = "Hello World!";
int len = i_only_know_c(i_am_modern_cpp.c_str()); // 12
```

Comparing std::string

Usually, the standard relational operators are used for string comparisons

- ==, <=> perform lexicographical comparisons
- Can only compare full strings

Example

```
std::string u0510 = "breezy badger";
std::string u1804 = "bionic beaver";
std::string u1904 = "disco dingo";

assert(u0510 == u0510); // obvious
assert(u1904 > u1804);  // okay, d after b
// three-way comparison: bi comes before br
assert((u1804 <=> u0510) == std::strong_ordering::less);
```

`std::string` Operations

The standard library provides additional operations on `std::string`

- `size()` or `length()`: The number of characters in the string
- `empty()`: Returns true if the string has no characters
- `append()` and `+=`: Appends another string or character. May incur memory allocations.
- Binary + concatenates two strings and returns a new heap-allocated string.
- `find()`: Returns the offset of the first occurrence of the substring, or the constant `std::string::npos` if not found
- `substr()`: Returns a new `std::string` that is a substring at the given offset and length. Be careful! Most of the times, you probably want a string view instead of a substring!

std::string_view (1)

Copying strings and creating substrings is expensive

- Whenever a substring is created, data is essentially duplicated
- Huge overhead when handling large amounts of data (e.g. parsing large JSON files)

`std::string_view` helps avoiding expensive copying

- Read-only views on already existing strings
- Internally: Just a pointer and a length
- Creation, substring and copying in constant time (vs. linear for strings)

`std::string_view` is defined in the `<string_view>` library header

- Creation: `std::string` (and optionally size) as constructor argument, from a char pointer with a length, or from a string literal
- Also has all (read-only) member functions of `std::string`
- Substring creates another string view in O(1)

Use `std::string_view` over `std::string` whenever possible!

std::string_view (2)

Example

```
std::string s = "garbage garbage garbage interesting garbage";

std::string sub = s.substr(24,11); // With string: O(n)

// With string view:
std::string_view s_view = s; // O(1)
std::string_view sub_view = s_view.substr(24,11); // O(1)

// Or in place:
s_view.remove_prefix(24); // O(1)
s_view.remove_suffix(s_view.size() - 11); // O(1)

// Also useful for function calls:
bool is_eq_naive(std::string a, std::string b) {return a == b; }
bool is_eq_views(std::string_view a, std::string_view b) {
     return a == b; }

is_eq_naive("abc", "def"); // 2 allocations at runtime
is_eq_with_views("abc", "def"); // no allocation at runtime
```

String Literals

Regular string literals do not handle null byte content correctly (see above)

- The standard library provides special literals ("suffixes") to construct `std::string_view` and `std::string` objects that deal with null bytes correctly.
- To use them, you have to use `using namespace std::literals::string_view_literals` or `using namespace std::literals::string_literals`.

Example

```
using namespace std::literals::string_view_literals;
using namespace std::literals::string_literals;

auto s1 = "string_view\0with\0nulls"sv; // s1 is a string_view
auto s2 = "string\0with\0nulls"s; // s2 is a string

std::cout << s1; // prints "string_viewwithnulls"
std::cout << s2; // prints "stringwithnulls"
```

Converting Numbers to Strings (std::to_chars)

The fastest way to convert numbers to strings is the `std::to_chars` function from the `<charconv>` header. Signature:

```
std::to_chars_result std::to_chars(char* first, char* last, T value);
```

- Can be used for any number type except `bool`
- Always uses `.` as decimal separator for floats
- Has more overloads to select different string representations
- Takes a range of a character buffer which will be written to
- Doesn't allocate memory!
- The return value has the two members `ec` (error code of type `std:errc`) and `ptr` (pointer past the end of the string that was written)

```
std::array<char, 10> buffer;
auto result = std::to_chars(
    buffer.data(), buffer.data() + buffer.size(), 1337);
assert(result.ec == std::errc{}); // No error occurred
std::string_view str(buffer.data(), result.ptr);
assert(str == "1337");
```

Converting Strings to Numbers (std::from_chars)

The fastest way to convert strings to numbers is the `std::from_chars` function from the `<charconv>` header. Signature:

```
std::from_chars_result std::from_chars(
    const char* first, const char* last, T& value);
```

- Similar semantics as `std::to_chars`
- Also has more overloads to parse different string formats
- String is read from a buffer range and result value is written to the `value` argument

```
std::string_view input = "1337";
int value;
auto result = std::from_chars(
    input.data(), input.data() + input.size(), value);
assert(result.ec == std::errc{}); // No error occurred
// Entire string was read
assert(result.ptr == input.data() + input.size());
assert(value == 1337);
```

std::optional (1)

Functions might fail or return without a valid result

- E.g. querying the size of a non-existent file
- We could naively try to encode such failures with a value of the function domain (e.g. zero size for non-existent files)
- Suboptimal, as there is no clear distinction between valid and invalid values

`std::optional` is a class encapsulating a value that might or might not exist

- Template class defined in the header `<optional>`
- Can either be empty, holding no value, or non-empty, holding an arbitrary value of its value type
- Provides a clean way to encode potentially missing values

std::optional (2)

Usage of `std::optional`

- `std::optional<T>`, where T can be almost any type (no references or arrays)
- Guarantees to not dynamically allocate any memory when being assigned a value
- Internally implemented as an object with a member that can store a T value and a boolean

Useful member functions

- `has_value()` or implicit conversion to `bool`: Check whether the optional contains a value
- Dereference operators `*` and `->`: Access or interact with the contained value (undefined behavior if the optional is empty)
- `value_or()`: Return the contained value if the optional is non-empty, or a default value otherwise
- `reset()`: Clear the optional

std::optional (3)

An optional is created through its constructor or with `std::make_optional`:

```
std::optional<std::string> might_fail(int arg) {
    if (arg == 0) {
        return std::optional<std::string>("zero");
    } else if (arg == 1) {
        return "one"; // equivalent to the case above
    } else if (arg < 7) {
        //std::make_optional takes constructor arguments of type T
        return std::make_optional<std::string>("less than 7");
    } else {
        return std::nullopt; // alternatively: return {}
    }
}
```

std::optional (4)

Checking the contents of an std::optional

```
might_fail(3).has_value(); // true
might_fail(8).has_value(); // false

// Or even simpler:
std::optional<std::string> opt5 = might_fail(5)
if (opt5) { //contextual conversion to bool
    opt5->size(); // 11
}
```

Accessing the value of an std::optional

```
*might_fail(3); // "less than 7"
// result will contain the string "less than 7", creates no copy
auto result = *might_fail(3);
might_fail(6)->size(); // 11
might_fail(7)->empty(); // undefined behavior
```

std::optional (5)

Providing a default value without boilerplate

```
might_fail(42).value_or("default"); // "default"
```

Clearing an optional

```
auto opt5 = might_fail(5)
opt5.has_value(); // true
opt5.reset(); // Clears the value
opt5.has_value(); // false
```

std::pair

std::pair<T, U> is a template class that stores exactly one object of type T and one of type U.

- Defined in the header <utility>
- Constructor takes object of T and U
- Pairs can also be constructed with std::make_pair()
- Objects can be accessed with first and second
- Can be compared for equality and inequality
- Can be compared lexicographically with ==, and <=>

```
std::pair<int, double> p1(123, 4.56);
p1.first; // == 123
p1.second; // == 4.56
auto p2 = std::make_pair(456, 1.23);
// p2 has type std::pair<double, int>
p1 < p2; // true
```

std::tuple

std::tuple is a template class with *n* type template parameters that stores exactly one object of each of the *n* types.

- Defined in the header `<tuple>`
- Constructor takes all objects
- Tuples can also be constructed with `std::make_tuple()`
- The *i*th object can be accessed with `std::get<i>()`
- Just like pairs, tuples define all relational comparison operators

```
std::tuple<int, double, char> t1(123, 4.56, 'x');
std::get<1>(t1); // == 4.56
auto p2 = std::make_tuple(456, 1.23, 'y');
// p2 has type std::tuple<int, double, char>
p1 < p2; // true
```

std::tie()

Tuples can also contain values of reference type. They can be constructed with std::tie().

- Can be used to easily “decompose” a tuple into existing variables
- Can also be used to quickly do lexicographic comparison on different objects

```
auto t = std::make_tuple(123, 4.56);
int a; double b;
std::tie(a, b) = t; // "decompose" t into a and b
// a is now 123, b is 4.56
int x = 456; double y = 1.23;
// Lexicographic comparison on (a, b) and (x, y):
std::tie(a, b) < std::tie(x, y); // true
```

Structured Bindings and Tuples

- Often, using structured bindings is easier than using `std::tie()`
- For tuples, `auto [a, b, c] = t;` initializes a, b, and c with `std::get<0>(t)`, `std::get<1>(t)`, and `std::get<2>(t)`, respectively
- Also works with `auto&` and `const auto&` in which case a, b, and c become references
- Also works with `std::pair`

```
auto t = std::make_tuple(1, 2, 3);
auto [a, b, c] = t; // a, b, c have type int
auto p = std::make_pair(4, 5);
auto& [x, y] = p; // x, y have type int&
x = 123; // p.first is now 123
```

Using Pairs and Tuples

`std::pair` and `std::tuple` should be used sparingly

- Convey no information about their intended semantics
- User-defined types can convey semantics through member names etc.
- User-defined types should almost always be preferred in public interfaces
- `std::pair` and `std::tuple` can be used internally

```
struct Rational {
    long numerator;
    long denominator;
};

std::pair<long, long> canonicalize(long, long); // BAD
Rational canonicalize(const Rational&);         // BETTER
```

Containers - A Short Overview

A container is an object that stores a collection of other objects

- Manage the storage space for their elements
- Generic: The type(s) of elements stored are template parameter(s)
- Provide member functions for accessing elements directly, or through iterators
- (Most) member functions shared between containers
- Make guarantees about the complexity of their operations:
  - Sequence containers (e.g. `std::array`, `std::vector`, `std::list`): Optimized for sequential access
  - Associative containers (e.g. `std::set`, `std::map`): Sorted, optimized for search ($O(\log n)$)
  - Unordered associative containers (e.g. `std::unordered_set`, `std::unordered_map`): Hashed, optimized for search (amortized: $O(1)$, worst case: $O(n)$)

Use containers whenever possible! When in doubt, use `std::vector`!

`std::vector`

Vectors are arrays that can dynamically grow in size

- Defined in the header `<vector>`
- Elements are still stored contiguously
- Elements can be inserted and removed at any position
- Preallocates memory for a certain amount of elements
- Allocates new, larger chunk of memory and moves elements when memory is exhausted
- Memory for a given amount of elements can be reserved with `reserve()`
- Time complexity:
  - Random access: $O(1)$
  - Insertion and removal at the end: Typically $O(1)$, worst case: $O(n)$ due to possible reallocation
  - Insertion and removal at any other position: $O(n)$
- Access to the underlying C-style data array with `data()` member function

std::vector<bool>

The class `std::vector<bool>` is an explicit specialization that works like a dynamic bitset.

- Individual values may not be stored contiguously (most likely one bit per value)
- Not possible to get pointers to elements
- No thread-safety guarantees for concurrent writes to different elements
- Most member functions exist and have the same complexity guarantees
- Should rarely be used because of its unusual properties

std::vector: Accessing Elements

Vectors are constructed just like arrays:

```
std::vector<int> fib = {1,1,2,3};
```

Access elements via array notation, or through a raw pointer:

```
fib[1] // == 1;

int* fib_ptr = fib.data();
fib_ptr[2] // == 3;
```

Update elements via array notation, or through a raw pointer:

```
fib[3] = 43;
fib[2] = 42;
fib.data()[1] = 41; // fib is now 1, 41, 42, 43
```

Note: It is not possible to insert new elements this way! You can only update existing ones.

std::vector: Inserting and Removing Elements

Insert or remove elements at the end in constant time:

```
fib.push_back(5); // fib is now 1, 1, 2, 3, 5
int my_fib = fib.back(); // my_fib is 5
fib.pop_back(); // fib is 1, 1, 2, 3
```

Insert or remove elements anywhere with an iterator pointing at the element after insertion, or the element to be erased respectively:

```
auto it = fib.begin(); it += 2;
fib.insert(it, 42); // fib is now 1, 1, 42, 2, 3

// insertion invalidated the iterator, get a new one
it = fib.begin(); it +=2;
fib.erase(it); // fib is now again 1, 1, 2, 3
```

Empty the whole vector with `clear`:

```
fib.clear();
fib.empty(); // true
fib.size(); // == 0
```

std::vector: Emplacing elements

Construct elements in place to avoid expensive moving around of data:

```
struct ExpensiveToCopy {
    ExpensiveToCopy(int id, std::string comment) :
        id(id), comment(std::move(comment)) {}
    int id;
    std::string comment;
};

std::vector<ExpensiveToCopy> vec;

// The expensive way:
ExpensiveToCopy e1(1,"my comment 1");
vec.push_back(e1); // need to copy e1!
// Better way, use std::move:
vec.push_back(std::move(e1));

// The best way:
vec.emplace_back(2, "my comment 2");

// Also works at any other position:
auto it = vec.begin(); it++;
vec.emplace(it, 3, "my comment 3");
```

std::vector: Reserving memory

If the final size of a vector is already known, give the vector a hint to avoid unnecessary reallocations:

```
std::vector<int> vec;
vec.reserve(1'000'000); //enough space for 1'000'000 elements is allocated
vec.capacity(); // == 1'000'000
vec.size(); // == 0, do not mix this up with capacity!

for (int i = 0; i < 1'000'000; ++i) {
    vec.push_back(i); // no reallocations in this loop!
}
```

std::span (1)

- References to individual objects can be passed around with pointers or references
- References to multiple objects that are stored contiguously could be passed around "manually" by using a pair of pointer and size
- Standard library abstracts this into the class `std::span<T>` in the header `<span>`
- Supports iteration, brackets operator, `data()`, `size()`
- Can be constructed from all contiguous containers (`std::array`, `std::vector`, C-Style array) and with pointer and size
- Subsets can be created with `subspan()`, no `T` objects are copied

Usage guidelines:

- Prefer using `std::span` over references to `std::array`, `std::vector`, etc.
- Use `std::span<const T>` if possible
- Pass `std::span` by copy in function arguments

std::span (2)

```
void printValues(std::span<const int> vs) {
    // Supports iteration
    for (auto v : vs) std::cout << v << '\n';
}

std::vector<int> values = {1, 2, 3, 4, 5};
std::span<int> valuesRef = values; // construct from container

valuesRef.size(); // == 5
valuesRef.data() == values.data(); // true
valuesRef[1]; // == 2

// Pass by copy (implicitly convert to span<const int>)
printValues(valuesRef);
// Create sub-span
printValues(valuesRef.subspan(2, 2)); // Prints 3, 4
```

std::unordered_map

Maps are associative containers consisting of key-value pairs

- Defined in the header `<unordered_map>`
- Keys are required to be unique
- At least two template parameters: `Key` and `T` (type of the values)
- Is internally a hash table
- Amortized $O(1)$ complexity for random access, search, insertion, and removal
- No way to access keys or values in order (use `std::map` for that!)
- Accepts custom hash- and comparison functions through third and fourth template parameter

Use `std::unordered_map` if you need a hash table and don't need ordering

std::unordered_map: Accessing Elements

Maps can be constructed pairwise:

```
std::unordered_map<std::string, double>
    name_to_grade {{"maier", 1.3}, {"huber", 2.7}, {"schmidt", 5.0}};
```

Lookup the value to a key with the brackets operator:

```
name_to_grade["huber"]; // == 2.7
```

A pair can also be searched for with `find`:

```
auto search = name_to_grade.find("schmidt");

if (search != name_to_grade.end()) {
    // Returns an iterator pointing to a pair!
    search->first; // == "schmidt"
    search->second; // == 5.0
}
```

To check if a key exists, use `contains`:

```
name_to_grade.contains("schmidt"); // true
name_to_grade.contains("blafasel"); // false
```

std::unordered_map: Insertion

Update or insert elements like this. If it did not exist, the brackets operator will insert a default-constructed value.
Note: The brackets operator has no const overload.

```
name_to_grade["moritz"]; // Entry {"moritz", 0.0} is inserted
// Entry {"michael", 0.0} is created, then value is set to 3.0
name_to_grade["michael"] = 3.0;
```

Maps also allow the direct insertion of pairs:

```
std::pair<std::string, double> pair("mueller", 1.0);
name_to_grade.insert(pair);

// Or simpler:
name_to_grade.insert({"mustermann", 3.7});

// Emplace also works:
name_to_grade.emplace("gruber", 1.7);
```

std::unordered_map: Removal

Erase elements with `erase()` or empty the container with `clear()`:

```
// Returns an iterator that points to the pair with "schmidt" as key
auto search = name_to_grade.find("schmidt");
// removes the element the iterator points to, returns iterator to next entry
auto newIterator = name_to_grade.erase(search);

// removes the pair with "moritz" as key, if it exists
size_t numRemoved = name_to_grade.erase("moritz");
// numRemoved is 1 if element was found and removed, 0 otherwise

name_to_grade.clear(); // removes all elements of name_to_grade
```

`std::map` (1)

In contrast to unordered maps, the keys of `std::map` are sorted

- Defined in the header `<map>`
- Interface largely the same to `std::unordered_map`
- Optionally accepts a custom comparison function as template parameter
- Is internally a tree (usually AVL- or R/B-Tree)
- $O(log\ n)$ complexity for random access, search, insertion, and removal

Use `std::map` only if you need a *sorted* associative container

std::map (2)

std:map also allows to search for ranges:

upper_bound() returns an iterator pointing to the first *greater* element:

```
std::map<int, int> x_to_y = {{1, 1}, {3, 9}, {7, 49}};
auto gt3 = x_to_y.upper_bound(3);

for (; gt3 != x_to_y.end(); ++gt3) {
    std::cout << gt3->first << "->" << gt3->second << ","; // 7->49,
}
```

lower_bound() returns an iterator pointing to the first element *not lower*:

```
auto geq3 = x_to_y.lower_bound(3);

for (; geq3 != x_to_y.end(); ++geq3) {
    std::cout << geq3->first << "->" << geq3->second << ","; // 3->9, 7->49,
}
```

std::unordered_set

Sets are associative containers consisting of keys

- Defined in the header `<unordered_set>`
- Keys are required to be unique (as is expected of a set)
- Template parameter `Key` for the type of the elements
- Is internally a hash table
- Amortized $O(1)$ complexity for random access, search, insertion, and removal
- No way to access keys in order (use `std::set` for that!)
- Elements must not be modified! If an element's hash changes, the container might get corrupted
- Accepts custom hash- and comparison functions through second and third template parameter

std::unordered_set: Checking for Elements

Sets can be constructed just like arrays:

```
std::unordered_set<std::string>
    shopping_list {"milk", "bread", "butter"};
```

Look for an element with find():

```
auto search = shopping_list.find("milk");

if (search != shopping_list.end()) {
    // Returns an iterator pointing to the element!
    *search; // == "milk"
}
```

Or with contains():

```
shopping_list.contains("bread"); // true
shopping_list.contains("blafasel"); // false
```

Check the number of the elements with size():

```
shopping_list.size(); // == 3
shopping_list.empty(); // false
```

std::unordered_set: Insertion

Update or insert elements like this:

```
shopping_list.insert("lettuce");

//Emplace also works:
shopping_list.emplace("milk");
```

`insert` returns a `std::pair<iterator,bool>` indicating if insertion succeeded:

```
auto result = shopping_list.insert("milk");

result.second; // false, as "milk" is already an element of shopping_list
*result.first; // "milk", iterator points to element preventing insertion

result = shopping_list.insert("broccoli");
result.second; // true, "broccoli" was added
*result.first; // "broccoli", iterator points to newly inserted element
```

std::unordered_set: Removal

Erase elements with `erase()` or empty it with `clear`:

```
// Returns an iterator that points to the "milk" element
auto search = shopping_list.find("milk");
// removes the element the iterator points to, returns iterator to next entry
auto newIterator = shopping_list.erase(search);

// removes the element "apples", if it exists
size_t numRemoved = name_to_grade.erase("apples");
// numRemoved is 1 if element was found and removed, 0 otherwise

shopping_list.clear(); // removes all elements of shopping_list
```

`std::set` (1)

In contrast to unordered sets, the elements of `std::set` are sorted

- Defined in the header `<set>`
- Interface largely the same to `std::unordered_set`
- Optionally accepts a custom comparison function as template parameter
- Is internally a tree (usually AVL- or R/B-Tree)
- $O(log\ n)$ complexity for random access, search, insertion, and removal

Use `std::set` only if you need a *sorted* set

std::set (2)

`std:set` also allows to search for ranges:
`upper_bound()` returns an iterator pointing to the first *greater* element:

```
std::set<int> x = {1, 3, 7};
auto gt3 = x.upper_bound(3);

for (; gt3 != x.end(); ++gt3) {
    std::cout << x << ","; // 7,
}
```

`lower_bound()` returns an iterator pointing to the first element *not lower*:

```
std::set<int> x = {1, 3, 7};
auto geq = x.lower_bound(3);

for (; geq != x.end(); ++geq) {
    std::cout << x << ","; // 3, 7,
}
```

Containers: Thread Safety

Containers give some thread safety guarantees:

- Two different containers: All member functions can be called concurrently by different threads (i.e. different containers don't share state)
- The same container: All read-only member functions can be called concurrently. E.g., `const` functions and `[]` (expect in associative containers), `data()`, `front()/back()`, `begin()/end()`, `find()`
- Iterator operations that only read (e.g. incrementing or dereferencing an iterator) can be run concurrently with reads of other iterators and `const` member functions
- Different elements of the same container can be modified concurrently
- Be careful: As long as the standard does not explicitly require a member function to be sequential, the standard library implementation is allowed to parallelize it internally (see e.g. `std::transform` vs. `std::for_each`)

Rule of thumb: Simultaneous reads on the same container are always okay, simultaneous read/writes on *different* containers are also okay. Everything else requires synchronization.

Iterators: A Short Overview

Iterators are objects that can be thought of as pointer abstractions

- Problem: Different element access methods for each container
- Therefore: Container types not easily exchangable in code
- Solution: Iterators abstract over element access and provide pointer-like interface
- Allow for easy exchange of underlying container type
- The standard library defines multiple iterator types as containers have varying capabilities (random access, traversable in both directions, ...)

Be careful: When writing to a container, all existing iterators are invalidated and can no longer be used (some exceptions apply)!

Iterators: An Example (1)

All containers have a begin and an end iterator:

```
std::vector<std::string> vec = {"one", "two", "three", "four"};
auto it = vec.begin();
auto end = vec.end();
```

The begin iterator points to the first element of the container:

```
std::cout << *it; // prints "one"
std::cout << it->size(); // prints 3
```

The end iterator points to the first element *after* the container. Dereferencing it results in undefined behavior:

```
*end; // undefined behavior
```

An iterator can be incremented (just like a pointer) to point at the next element:

```
++it; // Prefer to use pre-increment
std::cout << *it; // prints "two"
```

Iterators: An Example (2)

Iterators can be checked for equality. Comparing to the end iterator is used to check whether iteration is done:

```
// prints "three,four,"
for (; it != end; ++it) {
    std::cout << *it << ",";
}
```

This can be streamlined with a range-based for loop:

```
for (auto elem : vec) {
    std::cout << elem << ","; // prints "one,two,three,four,"
}
```

Such a loop requires the *range expression* (here: `vec`) to have a `begin()` and `end()` member.
`vec.begin()` is assigned to an internal iterator which is dereferenced, assigned to the *range declaration* (here: `auto elem`), and then incremented until it equals `vec.end()`.

Iterators: An Example (3)

Iterators can also simplify dynamic insertion and deletion:

```
for (it = vec.begin(); it != vec.end(); ++it) {
    if (it->size == 3) {
        it = vec.insert(it,"foo");
        // it now points to the newly inserted element
        ++it;
    }
}
//vec == {"foo", "one", "foo", "two", "three", "four"}

for (it = vec.begin(); it != vec.end(); ++it) {
    if (it->size == 3) {
        it = vec.erase(it);
        // erase returns a new, valid iterator
        // pointing at the next element
    }
}
//vec == {"three", "four"}
```

`input_iterator`, `output_iterator`

The standard library defines several concepts for different kinds of iterators in the `<iterator>` header. `std::input/output_iterator` are the most basic iterators. They have the following features:

- Equality comparison: Checks if two iterators point to the same position
- Dereferencable with the `*` and `->` operators
- Incrementable, to point at the next element in sequence
- A dereferenced `std::input_iterator` can *only* by read
- A dereferenced `std::output_iterator` can *only* be written to

As the most restrictive iterators, they have a few limitations:

- Single-pass only: They cannot be decremented
- Only allow equality comparison, <, >=, etc. not supported
- Can only be incremented by one (i.e. `it + 2` does *not* work)

Used in single-pass algorithms such as `find()` (`std::input_iterator`) or `copy()` (Copying from an `std::input_iterator` to an `std::output_iterator`)

forward_iterator, bidirectional_iterator

`std::forward_iterator` combines `std::input_iterator` and `std::output_iterator`

- All the features and restrictions shared between input- and output iterator apply
- Dereferenced iterator can be read and written to

`std::bidirectional_iterator` generalizes `std::forward_iterator`

- Additionally allows decrementing (walking backwards)
- Therefore supports multi-pass algorithms traversing the container multiple times
- All other restrictions of `std::forward_iterator` still apply

random_access_iterator, contiguous_iterator

`std::random_access_iterator` generalizes `std::bidirectional_iterator`

- Additionally allows random access with `operator[]`
- Supports relational operators, such as < or >=
- Can be incremented or decremented by any amount (i.e. `it + 2` *does* work)

`std::contiguous_iterator` generalizes `std::random_access_iterator`

- Guarantees that elements are stored in memory contiguously
- This means that iterators of this category can be used interchangeably with pointers: `&*(it + n) == (&*it) + n`

Streams and I/O

The standard library has an entire library for I/O operations. The main concept of the I/O library is a *stream*.

- Streams are organized in a class hierarchy
- `std::istream` is the base class for input operations (e.g. `operator>>`)
- `std::ostream` is the base class for output operations (e.g. `operator<<`)
- `std::iostream` is a subclass of `std::istream` and `std::ostream`
- `std::cin` is an instance of `std::istream` that represents stdin
- `std::cout` is an instance of `std::ostream` that represent stdout

As for strings, streams are actually templates parametrized with a character type.

- `std::istream` is an alias for `std::basic_istream<char>`
- `std::ostream` is an alias for `std::basic_ostream<char>`

Common Operations on Streams

All streams are subclasses of `std::basic_ios` and have the following member functions:

- `good()`, `fail()`, `bad()`: Checks if the stream is in a specific error state
- `eof()`: Checks if the stream has reached end-of-file
- `operator bool()`: Returns true if stream has no errors

```
int value;
if (std::cin >> value) {
    std::cout << "value = " << value << std::endl;
} else {
    std::cout << "error" << std::endl;
}
```

Input Streams

Input streams (`std::istream`) support several input functions:

- `operator>>()`: Reads a value of a given type from the stream, skips leading whitespace
- `operator>>()` can be overloaded for own types as second argument to support being read from a stream
- `get()`: Reads single or multiple characters until a delimiter is found
- `read()`: Reads given number of characters

```
// Defined by the standard library:
std::istream& operator>>(std::istream&, int&);
int value;
std::cin >> value;

// Read (up to) 1024 chars from cin:
std::vector<char> buffer(1024);
std::cin.read(buffer.data(), 1024);
```

Output Streams

Output streams (`std::ostream`) support several output functions:

- `operator<<()`: Writes a value to the stream
- `operator<<()` can be overloaded for own types as second argument to support being written to a stream
- `put()`: Writes a single character
- `write()`: Writes multiple characters

```
// Defined by the standard library:
std::ostream& operator<<(std::ostream&, int);
std::cout << 123;

// Write 1024 chars to cout:
std::vector<char> buffer(1024);
std::cout.write(buffer.data(), 1024);
```

String Streams

std::stringstream can be used when input and output should be written and read from a std::string.

- Defined in the header <sstream>
- Is a subclass of std::istream and std::ostream
- Initial contents can be given in the constructor
- Contents can be extracted and set with str()

```
std::stringstream stream("1 2 3");
int value;
stream >> value; // value == 1
stream.str("4"); // Set stream contents
stream >> value; // value == 4
stream << "foo";
stream << 123;
stream.str(); // == "foo123"
```

File Streams

The standard library defines several streams for file I/O in the <fstream> header:

- `std::ifstream`: Input file stream to read to a file
- `std::ofstream`: Output file stream to write to a file
- `std::fstream`: File stream to read and write to a file

```
std::ifstream input("input_file");
if (!input) { std::cout << "couldn't open input_file\n"; }
std::ofstream output("output_file");
if (!output) { std::cout << "couldn't open output_file\n"; }
// Read an int from input_file and write it to output_file
int value = -1;
if (!(input >> value)) {
    std::cout << "couldn't read from file\n";
}
if (!(output << value)) {
    std::cout << "couldn't write to file\n";
}
```

Disadvantage of Streams

Even though streams are nice to use, they should be avoided in many cases:

- Streams make heavy use of virtual functions and virtual inheritance which by itself can sometimes be a significant performance overhead
- Streams respect the system's locale settings (e.g. whether to use a period or a comma for floating point numbers) which also makes them slow
- Especially parsing of integers is very inefficient

General rule: When input is typed in by a user, using streams is fine. When input is read from files or generated automatically, better use OS-specific functions.

Standard Library II

Function Objects (1)

Regular functions are not objects in C++

- Cannot be passed as parameters
- Cannot have state
- ...

C++ additionally defines the *FunctionObject* named requirement. For a type `T` to be a *FunctionObject*

- `T` has to be an object
- `operator()(args)` has to be defined for `T` for a suitable argument list `args` which can be empty
- Often referred to as *functors*

Function Objects (2)

There are a number of valid function objects defined in C++

- Pointers to functions
- Lambda expressions
- Stateful function objects in form of classes

Functions and function references are not function objects

- Can still be used in the same way due to implicit function-to-pointer conversion

Function Pointers (1)

While functions are not objects they do have an address

- Location in memory where the actual assembly code resides
- Allows declaration of *function pointers*

Function pointers to non-member functions

- Declaration: `return-type (*identifier)(args)`
- Allows passing functions as parameters
  - E.g. passing a custom compare function to `std::sort` (see later)
  - E.g. passing a callback to a method
- Can be invoked in the same way as a function

Function Pointers (2)

Example

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
    return (*func)(arg1, arg2);
}
//--------------------------------------------------------
double callFunc(double (*func)(double), double argument) {
    return func(argument);  // Automatically dereferenced
}
//--------------------------------------------------------
int add(int arg1, int arg2) { return arg1 + arg2; }
double add4(double argument) { return argument + 4; }
//--------------------------------------------------------
int main() {
    auto i = callFunc(add, 2, 4); // i = 6
    auto j = callFunc(&add4, 4);  // j = 8, "&" can be omitted
}
```

Lambda Expressions (1)

Function pointers can be unwieldy

- Function pointers cannot easily capture environment
- Have to pass all variables that affect function by parameter
- Cannot have "local" functions within other functions

C++ defines *lambda expressions* as a more flexible alternative

- Lambda expressions construct a closure
- Closures store a function together with an environment
- Lambda expressions can *capture* variables from the scope where they are defined

Lambda Expressions (2)

Lambda expression syntax

- `[ captures ] ( params ) -> ret { body }`
- *captures* specifies the parts of the environment that should be stored
- *params* is a comma-separated list of function parameters
- *ret* specifies the return type and can be omitted, in which case the return type is deduced from return statements inside the body

The list of captures can be empty

- Results in stateless lambda expression
- Stateless lambda expressions are implicitly convertible to function pointers

Lambda expressions have unique unnamed class type

- This type cannot be named directly
- We have to rely on template argument deduction when assigning lambda expressions to variables (i.e. use `auto` or a deduced template parameter)

Lambda Expressions (3)

Example

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
    return func(arg1, arg2);
}
//--------------------------------------------------------
int main() {
    auto lambda = [](int arg1, int arg2) {
        return arg1 + arg2;
    };

    int i = callFunc(lambda, 2, 4);  // i = 6
    int j = lambda(5, 6);            // j = 11
}
```

Lambda Expressions (4)

All lambda expressions have *unique* types

```
// ERROR: Compilation will fail due to ambiguous return type
auto getFunction(bool first) {
    if (first) {
        return []() {
            return 42;
        };
    } else {
        return []() {
            return 42;
        };
    }
}
```

Lambda Captures (1)

Lambda captures specify what constitutes the state of a lambda expression

- Can refer to *automatic variables* in the surrounding scopes (up to the enclosing function)
- Can refer to the `this` pointer in the surrounding scope (if present)

Captures can either capture *by-copy* or *by-reference*

- Capture by-copy creates a copy of the captured variable in the lambda state
- Capture by-reference creates a reference to the captured variable in the lambda state
- Captures can be used in the lambda expression body like regular variables or references

Lambda Captures (2)

Lambda captures are provided as a comma-separated list of captures

- By-copy: *identifier* or *identifier initializer*
- By-reference: &*identifier* or &*identifier initializer*
- *identifier* must refer to automatic variables in the surrounding scopes
- *identifier* can be used as an identifier in the lambda body
- Each variable may be captured only once

First capture can optionally be a capture-default

- By-copy: =
- By-reference: &
- Allows any variable in the surrounding scopes to be used in the lambda body
- Specifies the capture type for all variables without explicit captures
- If present, only diverging capture types can be specified afterwards

Lambda Captures (3)

Capture types

```
int main() {
    int i = 0;
    int j = 42;

    auto lambda1 = [i](){};      // i by-copy
    auto lambda2 = [&i](){};     // i by-reference

    auto lambda2 = [&, i](){};  // j by-reference, i by-copy
    auto lambda3 = [=, &i](){}; // j by-copy, i by-reference

    auto lambda4 = [&, &i](){}; // ERROR: non-diverging capture types
    auto lambda5 = [=, i](){};  // ERROR: non-diverging capture types
}
```

Lambda Captures (4)

Capture by-copy vs. by-reference

```
int main() {
    int i = 42;

    auto lambda1 = [i]() { return i + 42; };
    auto lambda2 = [&i]() { return i + 42; };

    i = 0;

    int a = lambda1();  // a = 84
    int b = lambda2();  // b = 42
}
```

Lambda Captures (5)

We can also capture a this pointer

- By-copy: *this (actually copies the current object)
- By-reference: this

```
struct Foo {
    int i = 0;

    void bar() {
        auto lambda1 = [*this]() {return i + 42; };
        auto lambda2 = [this](){ return i + 42; };

        i = 42;

        int a = lambda1();  // a = 42
        int b = lambda2();  // b = 84
    }
};
```

Lambda Captures (6)

By-copy capture-default copies only the `this` pointer

```
struct Foo {
    int i = 0;

    void bar() {
        auto lambda1 = [&]() {return i + 42; };
        auto lambda2 = [=](){ return i + 42; };

        i = 42;

        int a = lambda1();  // a = 84
        int b = lambda2();  // b = 84
    }
};
```

Lambda Captures (7)

Beware of lifetimes when capturing

```
#include <memory>

int main() {
    auto ptr = std::make_unique<int>(4);

    auto f2 = [inner = ptr.get()]() {
        return *inner;
    };

    int a = f2();  // 4
    ptr.reset();
    int b = f2();  // undefined behavior
}
```

By-reference capture can also easily lead to dangling references

Stateful Function Objects (1)

Situation so far

- Functions are generally stateless
- State has to be kept in surrounding object, e.g. class instances
- Lambda expressions allow limited state-keeping

Function objects can be implemented in a regular class

- Allows the function object to keep arbitrary state
- Difference to lambda expressions: State member variables can be accessed explicitly and changed from outside the function object

Stateful Function Objects (2)

Example

```
struct Adder {
    int value;

    int operator()(int param) {
        return param + value;
    }
};
//--------------------------------------------------------
int main() {
    Adder myAdder;
    myAdder.value = 1;
    myAdder(1);          // 2
    myAdder(4);          // 5
    myAdder.value = 5;
    myAdder(1);          // 6
}
```

std::function (1)

std::function is a general purpose wrapper for all callable targets

- Defined in the <functional> header
- Able to store, copy and invoke the wrapped target
- Potentially incurs dynamic memory allocations
- Often adds unnecessary overhead
- Should be avoided where possible

```
#include <functional>
//------------------------------------------------------------
int add2(int p){ return p + 2; }
//------------------------------------------------------------
int main() {
    std::function<int(int)>adder = add2;
    int a = adder(5); // a = 7
}
```

std::function (2)

Potential std::function use case

```
#include <functional>
//------------------------------------------------------------
std::function<int()> getFunction(bool first){
    int a = 14;

    if (first)
        return [=]() { return a; };
    else
        return [=]() { return 2 * a; };
}
//------------------------------------------------------------
int main() {
    return getFunction(false)() + getFunction(true)(); // 42
}
```

Working with Function Objects

Code that intends to call function objects should usually rely on templates

```
int bad(int (*fn)()) { return fn(); }
//---------------------------------------------------------
template <typename Fn>
int good(Fn&& fn) { return fn(); }
//---------------------------------------------------------
struct Functor {
    int operator()() { return 42; }
};
//---------------------------------------------------------
int main() {
    Functor ftor;

    bad([]() { return 42; });   // OK
    bad(ftor);                  // ERROR

    good([]() { return 42; });  // OK
    good(ftor);                 // OK
}
```

The Algorithms Library

The algorithms library is part of the C++ standard library

- Defines operations on ranges of elements `[first, last)`
- Bundles functions for sorting, searching, manipulating, etc.
- Ranges can be specified using pointers or any appropriate iterator type
- Spread in 4 headers
  - `<algorithm>`
  - `<numeric>`
  - `<memory>`
  - `<cstdlib>`
- We will focus on `<algorithm>` as it bundles the most relevant parts

std::sort

Sorts all elements in a range `[first, last)` in ascending order

- `void sort(RandomIt first, RandomIt last);`
- Iterators must be `RandomAccessIterators`
- Elements have to be swappable (`std::swap` or user-defined swap)
- Elements have to be move-assignable and move-constructible
- Does not guarantee order of equal elements
- Needs $O(n * log(n))$ comparisons

```
#include <algorithm>
#include <vector>
//-------------------------------------------------------------
int main() {
    std::vector<unsigned> v = {3, 4, 1, 2};
    std::sort(v.begin(), v.end()); // 1, 2, 3, 4
}
```

Custom Comparison Functions

Sorting algorithms can be modified through custom comparison functions

- Supplied as function objects (`Compare` named requirement)
- Have to establish a strict weak ordering on the elements
- Syntax: `bool cmp(const Type1 &a, const Type2 &b);`
- Return `true` if and only if `a < b` according to some strict weak ordering `<`

```
#include <algorithm>
#include <vector>
//--------------------------------------------------------------
int main() {
    std::vector<unsigned> v = {3, 4, 1, 2};
    std::sort(v.begin(), v.end(), [](unsigned lhs, unsigned rhs) {
        return lhs > rhs;
    }); // 4, 3, 2, 1
}
```

Further Sorting Operations

Sometimes `std::sort` may not be the optimal choice

- Does not necessarily keep order of equal-ranked elements
- Sorts the entire range (unnecessary e.g. for top-k queries)

Keep the order of equal-ranked elements

- `std::stable_sort`

Partially sort a range

- `std::partial_sort`

Check if a range is sorted

- `std::is_sorted`
- `std::is_sorted_until`

Searching

The algorithms library offers a variety of searching operations

- Different set of operations for sorted and unsorted ranges
- Searching on sorted ranges is faster in general
- Sorting will pay off for repeated lookups

Arguments against sorting

- Externally prescribed order that may not be modified
- Frequent updates or insertions

General semantics

- Search operations return *iterators* pointing to the result
- Unsuccessful operations are usually indicated by returning the `last` iterator of a range `[first, last)`

Searching - Unsorted

Find the first element satisfying some criteria

- `std::find`
- `std::find_if`
- `std::find_if_not`

Search for a range of elements in another range of elements

- `std::search`

Count matching elements

- `std::count`
- `std::count_if`

Many more useful operations (see reference documentation)

std::find

Example

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------
int main() {
    std::vector<int> v = {2, 6, 1, 7, 3, 7};

    auto res1 = std::find(vec.begin(), vec.end(), 7);
    int a = std::distance(vec.begin(), res1); // 3

    auto res2 = std::find(vec.begin(), vec.end(), 9);
    assert(res2 == vec.end());
}
```

std::find_if

Example

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------------------
int main() {
    std::vector<int> v = {2, 6, 1, 7, 3, 7};

    auto res1 = std::find_if(vec.begin(), vec.end(),
        [](int val) { return (val % 2) == 1; });

    int a = std::distance(vec.begin(), res1); // 2

    auto res2 = std::find_if_not(vec.begin(), vec.end(),
        [](int val) { return val <= 7; });

    assert(res2 == vec.end());
}
```

Searching - Sorted

On sorted ranges, binary search operations are offered

- Complexity $O(log(n))$ when range is given as `RandomAccessIterator`
- Can employ custom comparison function (see above)

When called with `ForwardIterators` complexity is linear in number of iterator increments

Search for one occurrence of a certain element

- `std::binary_search`

Search for range boundaries

- `std::lower_bound`
- `std::upper_bound`

Search for all occurrences of a certain element

- `std::equal_range`

std::binary_search

Lookup an element in a range [first, last)

- Only checks for containment, therefore return type is bool
- To locate the actual values use std::equal_range

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::binary_search(v.begin(), v.end(), 3);
    assert(res1 == true);

    auto res2 = std::binary_search(v.begin(), v.end(), 0);
    assert(res2 == false);
}
```

std::lower_bound

Returns iterator pointing to the first element >= the search value

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::lower_bound(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), res1);  // 3

    auto res2 = std::lower_bound(v.begin(), v.end(), 0);
    int b = std::distance(v.begin(), res2);  // 0
}
```

std::upper_bound

Returns iterator pointing to the first element $>$ the search value

```
#include <algorithm>
#include <vector>
//-----------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::upper_bound(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), res1);  // 6

    auto res2 = std::upper_bound(v.begin(), v.end(), 4);
    assert(res2 == v.end());
}
```

std::equal_range

Locates range of elements equal to search value

- Returns pair of iterators (begin and end of range)
- Identical to using std::lower_bound and std::upper_bound

```
#include <algorithm>
#include <vector>
//--------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto [begin1, end1] = std::equal_range(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), begin1);  // 3
    int b = std::distance(v.begin(), end1);    // 6

    auto [begin2, end2] = std::equal_range(v.begin(), v.end(), 0);
    assert(begin2 == end2);
}
```

Permutations

The algorithms library offers operations to permute a given range

- Can iterate over permutations in lexicographical order
- Requires at least `BidirectionalIterators`
- Values have to be swappable
- Order is determined using `operator<` by default
- A custom comparison function can be supplied (see above)

Initialize a dense range of elements

- `std::iota`

Iterate over permutations in lexicographical order

- `std::next_permutation`
- `std::prev_permutation`

std::iota

Initialize a dense range of elements

- `std::iota(ForwardIt first, ForwardIt last, T value)`
- Requires at least `ForwardIterators`
- Fills the range `[first, last)` with increasing values starting at `value`
- Values are incremented using `operator++()`

```
#include <numeric>
#include <memory>
//--------------------------------------------------------
int main() {
    auto heapArray = std::make_unique<int[]>(5);
    std::iota(heapArray.get(), heapArray.get() + 5, 2);

    // heapArray is now {2, 3, 4, 5, 6}
}
```

std::next_permutation

Reorders elements in a range to the lexicographically next permutation

- `bool next_permutation(BidirIt first, BidirIt last)`
- Returns `false` if the current permutation was the lexicographically last permutation (the range is then sorted in descending order)

```
#include <algorithm>
#include <vector>
//-------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 3};

    bool b = std::next_permutation(v.begin(), v.end());
    // b == true, v == {1, 3, 2}
    b = std::next_permutation(v.begin(), v.end());
    // b == true, v == {2, 1, 3}
}
```

std::prev_permutation

Reorders elements in a range to the lexicographically previous permutation

- `bool prev_permutation(BidirIt first, BidirIt last)`
- Returns `false` if the current permutation was the lexicographically first permutation (the range is then sorted in ascending order)

```
#include <algorithm>
#include <vector>
//---------------------------------------------------------
int main() {
    std::vector<int> v = {1, 3, 2};

    bool b = std::prev_permutation(v.begin(), v.end());
    // b == true, v == {1, 2, 3}
    b = std::prev_permutation(v.begin(), v.end());
    // b == false, v == {3, 2, 1}
}
```

Additional Functionality

The algorithms library offers many more operations

- `std::min` & `std::max` over a range instead of two elements
- `std::merge` & `std::in_place_merge` for merging of sorted ranges
- Multiple set operations (intersection, union, difference, ...)
- Heap functionality
- Sampling of elements using `std::sample`
- Swapping elements using `std::swap`
- Range modifications
  - `std::copy` To copy elements to new location
  - `std::rotate` To rotate range
  - `std::shuffle` To randomly reorder elements
- For even more operations: See the reference documentation

The Ranges Library (1)

The ranges library provides components for dealing with ranges of elements

- Ranges provide an abstraction of the `[first, last)` iterator pairs we have seen so far
- Formalized by the `range` concept in the `<ranges>` header
- We can iterate over the elements of a range in the same way as over the elements of a container
- Newly introduced in C++20, compiler support may still be incomplete

Views of ranges can be manipulated through *range adaptors*

- Apply various transformations to a view or its contained elements
- Range adaptors can be composed in a functional way to yield more complex transformations

The Ranges Library (2)

Example

```
#include <ranges>
#include <iostream>
#include <map>

int main() {
    std::map<int, int> map{{1, 2}, {3, 4}};

    for (auto key : (map | std::views::keys))
        std::cerr << key << std::endl;
}
```

Output

```
1
3
```

Range Factories (1)

Most containers can directly be used as ranges

- Details specified by the range concepts in the `<range>` header
- In particular the `viewable_range` concept which allows a range to be converted to a `view` that can then be transformed further

Range *factories* can be used to create some commonly used views without constructing a dedicated container

- `views::empty` – An empty view
- `views::single` – A view that contains a single element
- `views::iota` – A view consisting of repeatedly incremented values

Range Factories (2)

Example

```
#include <ranges>
#include <iostream>

int main() {
    auto square = [](auto x) { return x * x; };

    for (auto i : (std::views::iota(1, 5)
                   | std::views::transform(square)))
        std::cout << i << std::endl;
}
```

Output

```
1
4
9
16
```

Range Adaptors (1)

Range adaptors apply transformations to the elements of a range

- Take a `viewable_range` as their first argument and return a `view`
- May take additional arguments if required by the transformation
- The pipe operator can be used to chain unary range adaptors

Chaining unary range adaptors

- Assume C_1 and C_2 to be unary range adaptors and `R` to be a range
- $C_2(C_1(R))$ is the view that results from applying C_1 followed by C_2 to `R`
- This can also be written as $R \mid C_1 \mid C_2$

Range Adaptors (2)

Example

```
#include <ranges>
#include <iostream>
#include <map>

int main() {
    std::map<int, int> map{{1, 2}, {3, 4}};

    // Functional syntax
    for (auto key : std::views::reverse(std::views::keys(map)))
        std::cerr << key << std::endl;

    // "Pipe" composition syntax
    for (auto key : (map | std::views::keys | std::views::reverse))
        std::cerr << key << std::endl;
}
```

Range Adaptors (3)

Range adaptors that take multiple arguments can be curried

- Assume `C` to be a range adaptor that takes arguments A_1 to A_n in addition to a range `R`
- Then `C(`A_1`, …, `A_n`)` is a unary range adaptor

This means the following expressions are equivalent

- `C(R, `A_1`, …, `A_n`)`
- `C(`A_1`, …, `A_n`)(R)`
- `R | C(`A_1`, …, `A_n`)`

Range Adaptors (4)

Example

```
#include <ranges>
#include <iostream>

int main() {
    auto numbers = {1, 2, 3, 4};
    auto square = [](auto x) { return x * x; };

    // Functional syntax
    for (auto i : std::views::transform(numbers, square))
        std::cout << i << std::endl;

    // Curried functional syntax
    for (auto i : std::views::transform(square)(numbers))
        std::cout << i << std::endl;

    // "Pipe" composition syntax
    for (auto i : numbers | std::views::transform(square))
        std::cout << i << std::endl;
}
```

Range Adaptors (5)

Many useful range adaptors are specified in the `<ranges>` header

- `views::filter` – View all elements that satisfy a predicate
- `views::transform` – Apply a transformation function to all elements
- `views::keys` – View the first elements of a range of pairs
- `views::values` – View the second elements of a range of pairs
- For more range adaptors see the reference documentation

The Random Library

The random library defines pseudo-random number generators and distributions

- Defined in `<random>` header
- Bundles several useful components
  - Abstraction for random devices
  - Random number generators
  - Wrappers to generate numerical distributions from RNGs

Should *always* be preferred over functionality from `<cstdlib>` header

- `rand` produces very low-quality random numbers
- E.g. in one example the lowest bit simply alternates between `0` and `1`
- Especially serious if `rand` is used with modulo operations

Random Number Generators (1)

The random library defines various pseudo-random number generators

- Uniform pseudo-random bit generators with distinct properties
- RNGs can be seeded and reseeded
- RNGs can be equality-compared
- RNGs are *not* thread-safe
- Within the STL, one should usually prefer the Mersenne Twister generators

The random library additionally defines a `default_random_engine` type alias

- Implementation is implementation-defined

Do not use if you want portability

Most RNGs are template specializations of an underlying random number *engine*

Always use the predefined RNGs unless you know **exactly** what you are doing

Random Number Generators (2)

Mersenne Twister engine

- Predefined for 32-bit (`std::mt19937`) and 64-bit (`std::mt19937_64`) output width
- Produces high-quality unsigned random numbers in $[0, 2^w - 1]$ where w is the number of bits
- Can and should be seeded in the constructor

```
#include <cstdint>
#include <random>
//------------------------------------------------------------
int main() {
    std::mt19937 engine(42);

    unsigned a = engine();  // a == 1608637542
    unsigned b = engine();  // b == 3421126067
}
```

std::random_device

Standard interface to every available source of external randomness

- /dev/random, atmospheric noise, ...
- Actual sources are implementation dependent
- Only “real” source of randomness

Can degrade to a pseudo-random number generator when no source of true randomness is available

```
#include <cstdint>
#include <random>
//--------------------------------------------------------
int main() {
    std::mt19937 engine(std::random_device()());

    unsigned a = engine();  // a == ???
    unsigned b = engine();  // b == ???
}
```

Distributions

Random number generators are rather limited

- Fixed output range
- Fixed output distribution (approximately uniform)

The random library provides *distributions* to transform the output of RNGs

- All distributions can be combined with all random engines
- Various well-known distributions are provided
  - Uniform
  - Normal
  - Bernoulli
  - Possion
  - ...
- Some distributions are available as discrete or continuous distributions

std::uniform_int_distribution

Generates discrete uniform random numbers in range $[a, b]$

- Integer type specified as template parameter
- Constructed as `uniform_int_distribution<T>(T a, T b)`
- If not specified a defaults to 0 and b to the maximum value of T
- Numbers generated by `operator()(Generator& g)` where g is any random number generator

```
#include <random>
//------------------------------------------------------------
int main() {
    std::mt19937 engine(42);
    std::uniform_int_distribution<int> dist(-2, 2);

    int d1 = dist(engine);  // d1 == -1
    int d2 = dist(engine);  // d2 == -2
}
```

std::uniform_real_distribution

Generates continuous uniform random numbers in range $[a, b]$

- Floating point type specified as template parameter
- Constructed as `uniform_real_distribution<T>(T a, T b)`
- If not specified a defaults to 0 and b to the maximum value of T
- Numbers generated by `operator()(Generator& g)` where g is any random number generator

```
#include <random>
//-------------------------------------------------------------
int main() {
    std::mt19937 engine(42);
    std::uniform_real_distribution<float> dist(-2, 2);

    float d1 = dist(engine);  // d1 == -0.50184
    float d2 = dist(engine);  // d2 == 1.18617
}
```

Seeding

Random generators should generate new random numbers each time

- The seed value of a generator is used to calculate all other random numbers
- Normally the seed should itself be a random number, e.g. by `random_device`
- Deterministic sequences are preferable e.g. for tests or experiments
- For tests or experiments seed can be fixed to an arbitrary integer

Entropy of a generator is entirely dependent on the entropy of the seed generator

Generating Random Dice Rolls

Example

```
#include <random>
//-----------------------------------------------------------
int main() {
    // Use random device to seed generator
    std::random_device rd;
    // Use pseudo-random generator to get random numbers
    std::mt19937 engine(rd());
    // Use distribution to generate dice rolls
    std::uniform_int_distribution<> dist(1, 6);

    int d1 = dist(engine); // gets random dice roll
    int d2 = dist(engine); // gets random dice roll
}
```

Problems With Modulo

Modulo should in general *not* be used to limit the range of RNGs

- Most random number generators generate values in $[0, 2^w - 1]$ for some w
- When using modulo with a number that is not a power of two modulo will favor smaller values

Consider e.g. random dice rolls

- Assume a perfect random generator gen with $w = 3$
- gen will produce all values in $\{0, \ldots, 7\}$ with equal probability 0.125

```
int randomDiceroll() {
    return gen() % 6 + 1;
}
```

- $P(randomDiceroll() = x) = 0.25$ for $x \in \{1, 2\}$
- $P(randomDiceroll() = x) = 0.125$ for $x \in \{3, 4, 5, 6\}$

Concurrency in Modern Hardware

Concurrency

What is concurrency?

```
function foo() { ... }
function bar() { ... }

function main() {
    t1 = startThread(foo)
    t2 = startThread(bar)

    // Wait for t1 and t2 to finish before continuing executing main()
    waitUntilFinished(t1)
    waitUntilFinished(t2)

    // No concurrent execution here anymore
}
```

In this example program, concurrency means that `foo()` and `bar()` are executed *at the same time*.

- How does a CPU actually do this?
- How can concurrency be used to make your programs faster?

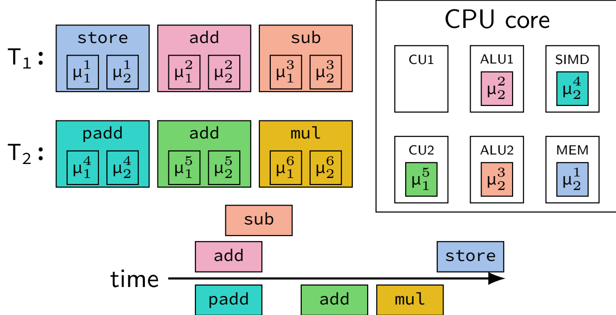
Concurrency in Modern Hardware

- Modern CPUs can execute multiple instruction streams simultaneously:
  - Single CPU cores can execute multiple threads: *Simultaneous Multi-Threading (SMT)*, Intel calls it hyper-threading
  - Of course CPUs can also have multiple cores that can run independently
- To get the best performance in C++ systems programming, writing multi-threaded programs is essential
- For this, a basic understanding of how hardware behaves in the context of parallel programming is required
- Actually writing multi-threaded C++ programs will be covered in a future lecture

Most of the low-level implementation details can be found in the Intel Architectures Software Developer's Manual and the ARM Architecture Reference Manual

Simultaneous Multi-Threading (SMT)

- CPUs support *instruction-level parallelism* by using out-of-order execution
- With SMT, CPUs also support *thread-level parallelism*
  - In a single CPU core, multiple threads are executed
  - Many hardware components, like the ALU, the SIMD unit, etc., are shared between the threads
  - Other components are duplicated for each thread, e.g. control unit to fetch and decode instructions, register file

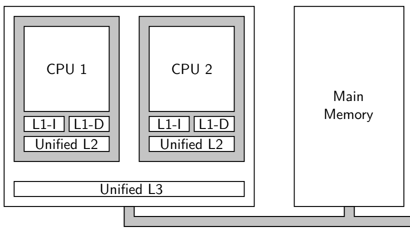


Problems with SMT

When using SMT, multiple instruction streams share parts of the CPU core.

- When one stream alone already utilizes all computation units, SMT does not increase performance
- Same for memory bandwidth
- Some units may only exist once on the core, so SMT can also decrease performance
- When two threads from unrelated processes run on the same core, this can potentially lead to security issues → Security issues similar to Spectre and Meltdown are suspected to be enabled by SMT

Cache Coherence

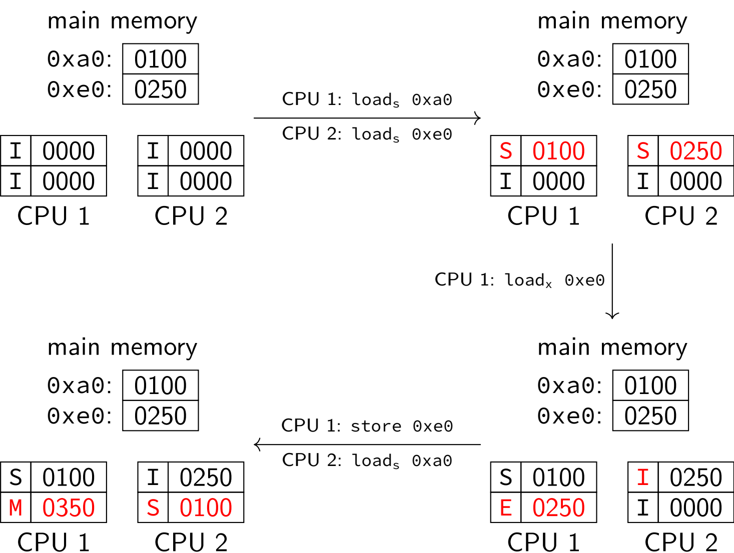


- Different cores can access the same memory at the same time
- Multiple cores potentially share caches
- Caches can be inclusive
- CPU must make sure that caching is consistent even with concurrent accesses → Communication between CPUs with a Cache Coherence Protocol

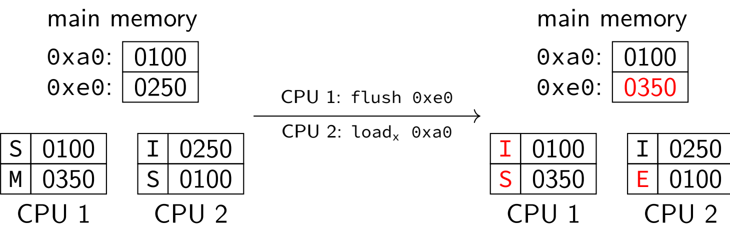
MESI Protocol

- CPUs and caches always read and write at cache line granularity, i.e. 64 byte
- The common MESI cache coherence protocol assigns every cache line one of the four states:
  - **M**odified: Cache line is stored in exactly one cache and was modified in the cache but not yet written back to main memory
  - **E**xclusive: Cache line is stored in exactly one cache to be used exclusively by one CPU
  - **S**hared: Cache line is stored in at least one cache, is currently used by a CPU for read-only access, and was not modified, yet
  - **I**nvalid: Cache line is not loaded or being used exclusively by another cache

MESI Example (1)



MESI Example (2)



- Memory store instructions don't directly write to main memory
- Can be enforced with explicit flush instructions
- When a cache line is used by multiple CPUs, this can lead to *cache thrashing*
  → bad performance

Memory Accesses and Concurrency

Consider the following example program where `foo()` and `bar()` will be executed concurrently:

```
globalCounter = 0

function foo() {
    repeat 1000 times:
        globalCounter = globalCounter - 1
}

function bar() {
    repeat 1000 times:
        globalCounter = globalCounter + 1
}
```

Machine code for this program could look like this:

```
foo:
  load (globalCounter), %r1
  sub %r1, $1
  store %r1, (globalCounter)
```

```
bar:
  load (globalCounter), %r1
  add %r1, $1
  store %r1, (globalCounter)
```

What is the value of `globalCounter` at the end?

Memory Order

- Out-of-order execution and simultaneous multi-processing leads to unexpected execution of memory load and store instructions
- All executed instructions will complete eventually
- However, effects of memory instructions (i.e. reads and writes) can become visible in a non-deterministic order
- CPU vendors define how reads and writes are allowed to be interleaved $\rightarrow$ *memory order*
- Generally: Dependent instructions within a single thread always work as expected:

```
store $123, A
load A, %r1
```

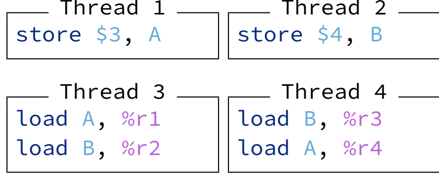
  If the memory location at `A` is only accessed by this thread, `r1` will always contain 123

Weak and Strong Memory Order

- CPU architectures usually have either *weak memory order* (e.g. ARM) or *strong memory order* (e.g. x86)
- Weak Memory Order:
  - As long as dependencies are respected, memory instructions and their effects can be reordered
  - Different threads will see writes in different orders
- Strong Memory Order:
  - Within a thread, only stores are allowed to be delayed after subsequent loads, everything else is not reordered
  - When two threads execute stores to the same location, all other threads will see the resulting writes in the same order
  - Writes from a set of threads will be seen in the same order by all other threads
- For both:
  - Writes from other threads can be reordered
  - Concurrent memory accesses to the same location can be reordered

Example of Memory Order (1)

In this example, initially the memory at A contains the value 1, the memory at B the value 2.



Weak memory order:

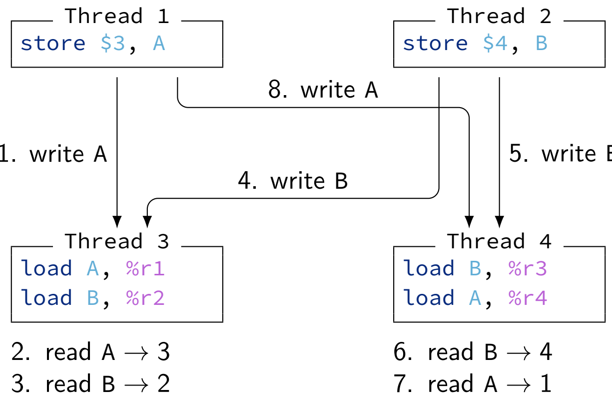
- Threads do not have dependent instructions
- Memory instructions can be reordered arbitrarily
- r1 $= 3$, r2 $= 2$, r3 $= 4$, r4 $= 1$ is allowed

Strong memory order:

- Threads 3 and 4 must see writes from threads 1 and 2 in the same order
- Example from weak memory order is not allowed
- r1 $= 3$, r2 $= 2$, r3 $= 4$, r4 $= 3$ is allowed

Example of Memory Order (2)

Visualization of the example for weak memory order:



- Thread 3 sees write to A (1.) before write to B. (4.)
- Thread 4 sees write to B (5.) before write to A. (8.)
- In strong memory order 5. is not allowed to happen before 8.

Memory Barriers

- Multi-core CPUs have special *memory barrier* (also called *memory fence*) instructions that can enforce stricter memory orders requirements
- This is especially useful for architectures with weak memory order
- x86 has the following barrier instructions:
  - `lfence`: Earlier loads cannot be reordered beyond this instruction, later loads and stores cannot be reordered before this instruction
  - `sfence`: Earlier stores cannot be reordered beyond this instruction, later stores cannot be reordered before this instruction
  - `mfence`: No loads or stores can be reordered beyond or before this instruction
- ARM has the *data memory barrier* instruction that supports different modes:
  - `dmb ishst`: All writes visible in or caused by this thread before this instruction will be visible to all threads before any writes from stores after this instruction
  - `dmb ish`: All writes visible in or caused by this thread and dependent reads before this instruction will be visible to all threads before any reads and writes after this instruction
- To additionally control out-of-order execution, ARM has the *data synchronization barrier* instructions: `dsb ishst`, `dsb ish`

Atomic Operations

- Memory order is only concerned about memory loads and stores
- Concurrent stores to the same memory location do not have any memory order constraints $\rightarrow$ order is possibly non-deterministic
- To allow deterministic concurrent modifications, most architectures support *atomic operations*
- An atomic operation is usually a sequence of: load data, modify data, store data
- Also called Read-Modify-Write (RMW)
- CPU ensures that all RMW operations are executed *atomically*, i.e. no other concurrent loads and stores are allowed in-between
- Usually only supported for individual arithmetic and bit-wise instructions

Atomic add on x86

```
lock addl $1, (%rdi)
```

Atomic add on ARM

```
ldrex   r1, [r0]
add     r1, r1, #1
strex   r2, r1, [r0]
```

Compare-And-Swap Operations (1)

- On x86, RMW instructions potentially lock the memory bus
- To avoid performance issues, only very few RMW instructions exist
- To facilitate more complex atomic operations, the *Compare-And-Swap* (CAS) atomic operation can be used
- ARM does not support locking the memory bus, so all RMW operations are implemented with CAS
- A CAS instruction has three parameters: The memory location `m`, the expected value `e`, and the desired value `d`
- The CAS operation conceptually works as follows:

```
tmp = load(m)
if (tmp == e) {
    store(m, d)
    success = true
} else {
    success = false
}
```

- Note: The CAS operation can fail, e.g. due to concurrent modifications!

Compare-And-Swap Operations (2)

Because CAS operations can fail, they are usually used in a loop with the following steps:

1. Load value from memory location into local register
2. Do computation with the local register assuming that no other thread will modify the memory location
3. Generate new desired value for the memory location
4. Do a CAS operation on the memory location with the value in the local register as expected value
5. Start the loop from the beginning if the CAS operation fails

Note that steps 2 and 3 can contain any number of instructions and are not limited to RMW instructions!

Compare-And-Swap Operations (3)

A typical loop using CAS looks like this:

```
success = false
while (not success) { (Step 5)
    expected = load(A) (Step 1)
    desired = non_trivial_operation(expected) (Steps 2, 3)
    success = CAS(A, expected, desired) (Step 4)
}
```

- With this approach, arbitrarily complex atomic operations on a memory location can be performed
- However, the likelihood for failure increases the more time is spent on the non-trivial operation
- Also, the non-trivial operation is potentially executed much more often than necessary

Parallel Programming

Parallel Programming

Multi-threaded programs usually contain many shared resources

- Data structures
- Operating system handles (e.g. file descriptors)
- Individual memory locations
- ...

Concurrent access to shared resources needs to be controlled

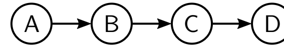
- Uncontrolled access leads to race conditions
- Race conditions usually end in inconsistent program state
- Other outcomes such as silent data corruption are also possible

Synchronization can be achieved in different ways

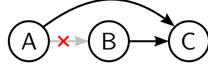
- Operating system support, e.g. through mutexes
- Hardware support, especially through atomic operations

Mutual Exclusion (1)

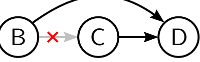
Concurrent removal of elements from a linked list



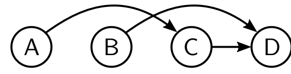
Thread 1 removes B



Thread 2 removes C



Final state



Observations

- C is not actually removed
- Threads might also deallocate node memory after removal

Mutual Exclusion (2)

Protect shared resources by only allowing accesses within critical sections

- Only one thread at a time can enter a critical section
- Ensures that the program state is always consistent if used correctly
- Non-deterministic (but consistent) program behavior is still possible

There are various possibilities for implementing mutual exclusion

- Atomic test-and-set operations
  - usually requires spinning which can be dangerous
- Operating system support
  - E.g. mutexes in Linux

Locks

Implement mutual exclusion by acquiring locks on mutex objects

- Only one thread at a time can acquire a lock on a mutex
- Trying to acquire a lock on an already locked mutex will block the thread until the mutex becomes available again
- Blocked threads can be suspended by the kernel to free compute resources

Multiple mutex objects can be used to represent separate critical sections

- Only one thread at a time may enter the same critical section, but threads may simultaneously enter distinct critical sections
- Allows for more fine-grained synchronization
- Requires careful implementation to avoid deadlocks

Shared Locks

Strict mutual exclusion is not always necessary

- Commonly concurrent read-only accesses to the same shared resource do not interfere with each other
- Using strict mutual exclusion introduces an unnecessary bottleneck as readers would block each other
- We only need to make sure that write accesses can not happen concurrently with other write or read accesses

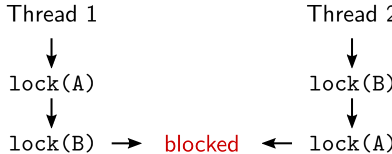
Shared locks provide a solution

- Threads can acquire either an exclusive or a shared lock on a mutex
- Multiple threads can simultaneously acquire a shared lock on a mutex if it is not locked exclusively
- One thread at a time can acquire an exclusive lock on a mutex if it is not locked in any other way (exclusive or shared)

Problems with Mutual Exclusion (1)

Deadlocks

- Multiple threads each wait for the other threads to release a lock



Avoiding deadlocks

- If possible, threads should never acquire multiple locks
- If not avoidable, locks must always be acquired in a globally consistent order

Problems with Mutual Exclusion (2)

Starvation

- High contention on a mutex may lead to some threads making no progress
- Can partially be alleviated by using less restrictive locking schemes

High latency

- Some threads are blocked for a long time if a mutex is highly contended
- Can lead to noticeably reduced system performance
- Performance can possibly even drop below single-threaded performance

Priority inversion

- A high-priority thread may be blocked by a low-priority thread
- Due to the priority differential, the low-priority thread may not be allowed sufficient compute resources to quickly release the lock

Hardware-Assisted Synchronization

Using mutexes is usually relatively expensive

- Each mutex requires some state (16 to 40 bytes)
- Acquiring locks potentially requires system calls which can take thousands of cycles or more

For this reason, mutexes are best suited for coarse-grained locking

- E.g. locking an entire data structure instead of parts of it
- Sufficient if only very few threads contend for locks on the mutex
- Sufficient if the critical section protected by the mutex is much more expensive than a (potential) system call to acquire a lock

The performance of mutexes quickly degrades under high contention

- In particular, the latency of lock acquisition increases dramatically
- This even occurs when we only acquire shared locks on a mutex
- We can exploit hardware support for more efficient synchronization

Optimistic Locking (1)

Often, read-only accesses to a resource are more common than write accesses

- Thus we should optimize for the common case of read-only access
- In particular, parallel read-only access by many threads should be efficient
- Shared locks are not well-suited for this (see previous slide)

Optimistic locking can provide efficient reader-writer synchronization

- Associate a *version* with the shared resource
- Writers still have to acquire an exclusive lock of some sort
  - This ensures that only one writer at a time has access to the resource
  - At the end of its critical section, a writer atomically increases the version
- Readers only have to read the version
  - At the begin of its critical section, a reader atomically reads the current version
  - At the end of its critical section, a reader validates that the version did not change
  - Otherwise, a concurrent write occurred and the critical section is restarted

Optimistic Locking (2)

Example (pseudocode)

```
writer(optLock) {
    lockExclusive(optLock.mutex) // begin critical section

    // modify the shared resource

    storeAtomic(optLock.version, optLock.version + 1)

    unlockExclusive(optLock.mutex) // end critical section
}

reader(optLock) {
    while(true) {
        current = loadAtomic(optLock.version); // begin critical section

        // read the shared resource

        if (current == loadAtomic(optLock.version)) // validate
            return; // end critical section
    }
}
```

Optimistic Locking (3)

Why is optimistic locking efficient?

- Readers only have to execute two atomic load instructions
- This is much cheaper than acquiring a shared lock
- But requires that modifications are rare, otherwise readers have to restart frequently

A careful implementation of readers is required

- The shared resource may be modified while a reader is accessing it
- We cannot assume that we read from a consistent state
- Additional intermediate validation may be required for more complex read operations

Beyond Mutual Exclusion

In many cases, strict mutual exclusion is not required in the first place

- E.g. parallel insertion into a linked list
- We do not care about the order of insertions
- We only need to guarantee that all insertions are reflected in the final state

This can be implemented efficiently by using atomic operations (pseudocode)

```
threadSafePush(linkedList, element) {
    while (true) {
        head = loadAtomic(linkedList.head)
        element.next = head
        if (CAS(linkedList.head, head, element))
            break;
    }
}
```

Non-Blocking Algorithms

Algorithms or data structures that do not rely on locks are called *non-blocking*

- E.g. the `threadSafePush` function on the previous slide
- Synchronization between threads is usually achieved using atomic operations
- Enables more efficient implementations of many common algorithms and data structures

Such algorithms can provide different levels of progress guarantee

- Wait-freedom: There is an upper bound on the number of steps it takes to complete each operation
  - Hard to achieve in practice
- Lock-freedom: At least one thread makes progress if the program is run for sufficient time
  - Often informally (and technically incorrectly) used as a synonym for non-blocking

A-B-A Problem (1)

Non-blocking data structures need to be implemented carefully

- We do not have the luxury of critical sections anymore
- Threads can execute different operations on a data structure in parallel (e.g. insert and remove)
- The individual atomic operations comprising these compound operations can be interleaved arbitrarily
- This can lead to hard-to-debug anomalies, such as lost updates or the A-B-A problem

Often problems can be avoided by making sure that only the same operation (e.g. insert) is executed in parallel

- E.g. insert elements in parallel in a first step, and remove them in parallel in a second step

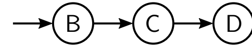
A-B-A Problem (2)

Consider the following simple linked-list based stack (pseudocode)

```
threadSafePush(stack, element) {
    while (true) {
        head = loadAtomic(stack.head)
        element.next = head
        if (CAS(stack.head, head, element))
            break;
    }
}

threadSafePop(stack) {
    while (true) {
        head = loadAtomic(stack.head)
        next = head.next
        if (CAS(stack.head, head, next))
            return head
    }
}
```

A-B-A Problem (3)

Consider the following initial state of the stack, on which two threads perform some operations in parallel



Thread 1

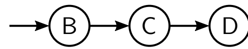
```
x = threadSafePop(stack)
```

Thread 2

```
y = threadSafePop(stack)
z = threadSafePop(stack)
threadSafePush(stack, y)
```

A-B-A Problem (4)

Our implementation would allow the execution to be interleaved as follows

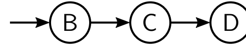


Thread 1

Thread 2

A-B-A Problem (5)

Our implementation would allow the execution to be interleaved as follows



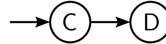
Thread 1

```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

A-B-A Problem (6)

Our implementation would allow the execution to be interleaved as follows



Thread 1

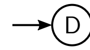
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
```

A-B-A Problem (7)

Our implementation would allow the execution to be interleaved as follows



Thread 1

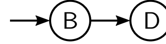
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
```

A-B-A Problem (8)

Our implementation would allow the execution to be interleaved as follows



Thread 1

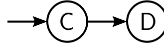
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
threadSafePush(stack, y)
```

A-B-A Problem (9)

Our implementation would allow the execution to be interleaved as follows



Thread 1

```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C




CAS(stack.head, head, next)
// inconsistent state!
```

Thread 2

```




y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
threadSafePush(stack, y)
```

The Dangers of Spinning (1)

It is possible to implement a “better” mutex that requires less space and uses no system calls by using atomic operations:

- The mutex is represented in a single atomic integer
- It has the value 0 when it is unlocked, 1 when it is locked
- To lock the mutex, the value is changed atomically to 1 only if it was 0 by using a CAS
- The CAS is repeated as long as another thread holds the mutex

```
function lock(mutexAddress) {
    while (CAS(mutexAddress, 0, 1) not sucessful) {
        <noop>
    }
}

function unlock(mutexAddress) {
    atomicStore(mutexAddress, 0)
}
```

The Dangers of Spinning (2)

Using this CAS loop as a mutex, also called *spin lock*, has several disadvantages:

- It has no fairness, i.e. does not guarantee that a thread will acquire the lock eventually → *starvation*
- The CAS loop consumes CPU cycles (waste of energy and resources)
- Can easily lead to *priority inversion*
  - The scheduler of the operating system thinks that the spinning thread requires a lot of CPU time
  - The spinning thread actually does no useful work at all
  - In the worst-case, the scheduler takes CPU time away from the thread that holds the lock to give it to the spinning thread

  → Spinning thread needs to spin even longer which makes the situation worse

Possible solutions:

- Spin for a limited number of times (e.g. several hundred thousand iterations)
- If the lock could not be acquired, fall back to a “real” mutex
- This is actually already how mutexes are usually implemented

Multi-Threading in C++

Multi-Threading in C++

In C++ it is allowed to run multiple threads simultaneously that use the same memory.

- Multiple threads may *read* from the same memory location
- All other accesses (i.e. read-write, write-read, write-write) are called *conflicts*
- Conflicting operations are only allowed when threads are *synchronized*
- This can be done with *mutexes* or *atomic operations*
- Unsynchronized accesses (also called *data races*), deadlocks, and other potential issues when using threads are undefined behavior!

All conflicting operations must be synchronized in some way!

Threads Library (1)

The header <thread> defines the class `std::thread`

- Using this class is the best way to use threads platform-independently
- May require additional compiler flags depending on the actual underlying implementation
- Use CMake to determine these flags in a platform-independent way
- For `gcc` and `clang` on Linux this will usually be `-pthread`

```
cmake_minimum_required(VERSION 3.21)
project(sample)

find_package(Threads REQUIRED)
add_executable(sample main.cpp)
target_link_libraries(sample PUBLIC Threads::Threads)
```

Threads Library (2)

The constructor of `std::thread` can be used to start a new thread

- Syntax: `thread(Function&& f, Args&&... args)`
- The function `f` will be invoked in a new thread with the arguments `args`
- The thread will terminate once `f` returns
- The default constructor can be used to create an empty thread object

The member function `join()` must be used to wait for a thread to finish

- `join()` must be called exactly once for each thread
- `join()` must be called *before* an `std::thread` object is destroyed
- When the destructor of an `std::thread` is called, the program is terminated if the associated thread was not joined

Threads Library (3)

Example

```
#include <thread>

void foo(int a, int b);

int main() {
    // Pass a function and args
    std::thread t1(foo, 1, 2);
    // Pass a lambda
    std::thread t2([]() {
        foo(3, 4);
    });

    foo(5, 6);

    t2.join();
    t1.join();
}
```

Threads Library (4)

Example

```
#include <iostream>
#include <string_view>
#include <thread>

void safe_print(std::string_view s);

int main() {
    {
        std::thread t1([]() { safe_print("Hi\n"); });
        t1.join();
    }
    // Everything is fine, we called t1.join()
    {
        std::thread t2([]() {});
    }
    // Program terminated because t2.join() was not called
}
```

Threads Library (5)

std::thread is movable but not copyable

- Moving transfers all resources associated with the running thread
- Only the moved-to thread can be joined
- The moved-from thread object is empty (not associated with any thread)

Example

```
#include <iostream>
#include <string_view>
#include <thread>

void safe_print(std::string_view s);

int main() {
    std::thread t1([]() { safe_print("Hi\n"); });
    std::thread t2 = std::move(t1); // t1 is now empty
    t2.join(); // OK, thread originally started in t1 is joined
}
```

Threads Library (6)

std::thread can be used in standard library containers

```
#include <thread>
#include <vector>

void safe_print(int i);

int main() {
    std::vector<std::thread> threadPool;
    for (int i = 1; i <= 9; ++i) {
        threadPool.emplace_back([i]() { safe_print(i); });
    }
    // Digits 1 to 9 are printed (unordered)
    for (auto& t : threadPool) {
        t.join();
    }
}
```

Other Functions of the Thread Library

The thread library also contains other useful functions that are closely related to starting and stopping threads:

- `std::this_thread::sleep_for()`: Stop the current thread for a given amount of time
- `std::this_thread::sleep_until()`: Stop the current thread until a given point in time
- `std::this_thread::yield()`: Let the operating system schedule another thread
- `std::this_thread::get_id()`: Get the (operating-system-specific) id of the current thread

Mutual Exclusion (1)

Mutual exclusion is a straightforward way to synchronize multiple threads

- Threads acquire a lock on a mutex object before entering a critical section
- Threads release their lock on the mutex when leaving a critical section

High-level programming model

- The resource (usually a class) that requires protection from data races owns a mutex object of the appropriate type
- Threads that intend to access the resource acquire a suitable lock on the mutex *before* performing the actual access
- Threads release their lock on the mutex *after* completing the access
- Usually locks are simply acquired and released in the member functions of the class

Mutual Exclusion (2)

The standard library defines several useful classes that implement mutexes in the `<mutex>` and `<shared_mutex>` headers

- `std::mutex` – regular mutual exclusion
- `std::recursive_mutex` – recursive mutual exclusion
- `std::shared_mutex` – mutual exclusion with shared locks

The standard library provides RAII wrappers for locking and unlocking mutexes

- `std::unique_lock` – RAII wrapper for exclusive locking
- `std::shared_lock` – RAII wrapper for shared locking

The RAII wrappers should *always* be preferred for locking and unlocking mutexes

- Makes bugs due to inconsistent locking/unlocking much more unlikely
- Manual locking and unlocking may be required in some rare cases
- Should still be performed through the corresponding functions of the RAII wrappers

std::unique_lock (1)

std::unique_lock can be used to lock a mutex in exclusive mode

- The constructor acquires an exclusive lock on the mutex
- Constructor syntax: unique_lock(mutex_type& m)
- Blocks the calling thread until the mutex becomes available
- The destructor releases the lock automatically
- Can be used with any mutex type from the standard library

```
#include <mutex>
#include <iostream>

std::mutex printMutex;
void safe_print(int i) {
    std::unique_lock lock(printMutex); // lock is acquired
    std::cout << i;
} // lock is released
```

std::unique_lock (2)

std::unique_lock provides additional constructors

- unique_lock(mutex_type& m, std::defer_lock_t t) – Do not immediately lock the mutex
- unique_lock(mutex_type& m, std::try_to_lock_t t) – Do not block when the mutex cannot be locked

std::unique_lock provides additional member functions

- lock() – Manually lock the mutex
- try_lock() – Try to lock the mutex, return true if successful
- operator bool() – Check if the std::unique_lock holds a lock on the mutex

std::unique_lock (3)

Example

```
#include <mutex>

std::mutex mutex;

void foo() {
    std::unique_lock lock(mutex, std::try_to_lock);
    if (!lock) {
        doUnsynchronizedWork();

        // block until we can get the lock
        lock.lock();
    }

    doSynchronizedWork();

    // release the lock early
    lock.unlock();

    doUnsynchronizedWork();
}
```

std::unique_lock (4)

std::unique_lock is movable to transfer ownership of a lock on a mutex

```
#include <mutex>

class MyContainer {
    private:
    std::mutex mutex;

    public:
    class iterator { /* ... */ };

    iterator begin() {
        std::unique_lock lock(mutex);

        // compute the begin iterator constructor args

        // keep the lock for iteration
        return iterator(std::move(lock), ...);
    }
};
```

Recursive Mutexes (1)

The following code will deadlock since `std::mutex` can be locked at most once

```
#include <mutex>

std::mutex mutex;

void bar() {
    std::unique_lock lock(mutex);

    // do some work...
}

void foo() {
    std::unique_lock lock(mutex);

    // do some work...

    bar(); // INTENTIONALLY BUGGY, will deadlock
}
```

Recursive Mutexes (2)

`std::recursive_mutex` implements recursive ownership semantics

- The same thread can lock an `std::recursive_mutex` multiple times without blocking
- Other threads will still block if an `std::recursive_mutex` is currently locked
- Can be used with `std::unique_lock` just like a regular `std::mutex`
- Useful for functions that call each other and use the same mutex

```
#include <mutex>

std::recursive_mutex mutex;
void bar() {
    std::unique_lock lock(mutex);
}
void foo() {
    std::unique_lock lock(mutex);
    bar(); // OK, will not deadlock
}
```

std::shared_lock (1)

std::shared_lock can be used to lock a mutex in shared mode

- Constructors and member functions analogous to std::unique_lock
- Multiple threads can acquire a shared lock on the same mutex
- Shared locking attempts block if the mutex is locked in exclusive mode
- Only usable in conjunction with std::shared_mutex

We have to adhere to some contract to write well-behaved programs

- Shared mutexes are mostly used to implement read/write-locks
- Only read accesses are allowed when holding a shared lock
- Write accesses are only allowed when holding an exclusive lock

std::shared_lock (2)

Example

```
#include <shared_mutex>

class SafeCounter {
    private:
    mutable std::shared_mutex mutex;
    size_t value = 0;

    public:
    size_t getValue() const {
        std::shared_lock lock(mutex);
        return value; // read access
    }

    void incrementValue() {
        std::unique_lock lock(mutex);
        ++value; // write access
    }
};
```

Working with Mutexes

We usually have to make mutexes `mutable` within our data structures

- The RAII wrappers require mutable references to the mutex
- `const` member functions of our data structure usually also need to use the mutex

Using mutexes without care can easily lead to deadlocks within the system

- Usually occurs when a thread tries to lock another mutex when it already holds a lock on some mutex
- Can in some cases be avoided by using `std::recursive_mutex` (if we are locking the same mutex multiple times)
- Requires dedicated programming techniques when multiple mutexes are involved

Avoiding Deadlocks (1)

The following example will lead to deadlocks

```
std::mutex m1, m2, m3;
void threadA() {
    // INTENTIONALLY BUGGY
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    // INTENTIONALLY BUGGY
    std::unique_lock l3{m3}, l2{m2}, l1{m1};
}
```

Possible deadlock scenario

- `threadA()` acquires locks on `m1` and `m2`
- `threadB()` acquires lock on `m3`
- `threadA()` waits for `threadB()` to release `m3`
- `threadB()` waits for `threadA()` to release `m2`

Avoiding Deadlocks (2)

Deadlocks can be avoided by always locking mutexes in a *globally* consistent order

- Ensures that one thread always "wins"
- Maintaining a globally consistent locking order requires considerable developer discipline
- Maintaining a globally consistent locking order may not be possible at all

```
std::mutex m1, m2, m3;
void threadA() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
```

Avoiding Deadlocks (3)

Sometimes it is not possible to guarantee a globally consistent order

- The `std::scoped_lock` RAII wrapper can be used to safely lock any number of mutexes
- Employs a deadlock-avoidance algorithm if required
- Generally quite inefficient in comparison to `std::unique_lock`
- Should only be used as a last resort!

```
std::mutex m1, m2, m3;
void threadA() {
    // OK, will not deadlock
    std::scoped_lock l{m1, m2, m3};
}
void threadB() {
    // OK, will not deadlock
    std::scoped_lock l{m3, m2, m1};
}
```

Condition Variables (1)

A condition variable is a synchronization primitive that allows multiple threads to wait until an (arbitrary) condition becomes true.

- A condition variable uses a mutex to synchronize threads
- Threads can *wait* on or *notify* the condition variable
- When a thread waits on the condition variable, it blocks until another thread notifies it
- If a thread waited on the condition variable and is notified, it holds the mutex
- A notified thread must check the condition explicitly because *spurious wake-ups* can occur

Condition Variables (2)

The standard library defines the class `std::condition_variable` in the header `<condition_variable>` which has the following member functions:

- `wait()`: Takes a reference to a `std::unique_lock` that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable
- `notify_one()`: Notify a single waiting thread, mutex does not need to be held by the caller
- `notify_all()`: Notify all waiting threads, mutex does not need to be held by the caller

Condition Variables Example

One use case for condition variables are worker queues: Tasks are inserted into a queue and then worker threads are notified to do the task.

```
std::mutex m;
std::condition_variable cv;
std::vector<int> taskQueue;

void pushWork(int task) {
  {
    std::unique_lock l{m};
    taskQueue.push_back(task);
  }
  cv.notify_one();
}
```

```
void workerThread() {
  std::unique_lock l{m};
  while (true) {
    while (!taskQueue.empty()) {
      int task = taskQueue.back();
      taskQueue.pop_back();
      l.unlock();
      // [...] do actual work here
      l.lock();
    }
    cv.wait(l);
  }
}
```

Atomic Operations

Mutual exclusion may be inefficient for synchronization

- Very coarse-grained synchronization
- May require communication with the operating system

Modern hardware also supports *atomic operations* for synchronization.

- The memory order of a CPU determines how *non-atomic* memory operations are allowed to be reordered
- In C++ all non-atomic conflicting operations have undefined behavior even if the memory order of the CPU would allow it!
- There is one exception: Special atomic functions are allowed to have conflicts
- The compiler usually knows your CPU and generates "real" atomic instructions only if necessary

Atomic Operations Library (1)

C++ provides atomic operations in the atomic operations library

- Implemented in the `<atomic>` header
- `std::atomic<T>` is a class that represents an atomic version of the type `T`
- Can be used (almost) interchangeably with the original type `T`
- Has the same size and alignment as the original type `T`
- Conflicting operations are only allowed on `std::atomic<T>` objects

`std::atomic` on its own does not provide any synchronization at all

- Simply makes conflicting operations possible and defined behavior
- Exposes the guarantees of specific memory models to the programmer
- Suitable programming models must be used to achieve proper synchronization

Atomic Operations Library (2)

`std::atomic` has several member functions that implement atomic operations

- `T load()`: Loads the value
- `void store(T desired)`: Stores `desired` in the object
- `T exchange(T desired)`: Stores `desired` in the object and returns the old value

If `T` is a integral type, the following operations also exist:

- `T fetch_add(T arg)`: Adds `arg` to the value and returns the old value
- `T fetch_sub(T arg)`: Same for subtraction
- `T fetch_and(T arg)`: Same for bitwise and
- `T fetch_or(T arg)`: Same for bitwise or
- `T fetch_xor(T arg)`: Same for bitwise xor

Atomic Operations Library (3)

Example (without atomics)

```
#include <thread>

int main() {
    unsigned value = 0;
    std::thread t([&]() {
        for (size_t i = 0; i < 10; ++i)
            ++value; // UNDEFINED BEHAVIOR, data race
    });

    for (size_t i = 0; i < 10; ++i)
        ++value; // UNDEFINED BEHAVIOR, data race

    t.join();

    // value will contain garbage
}
```

Atomic Operations Library (4)

Example (with atomics)

```
#include <atomic>
#include <thread>

int main() {
    std::atomic<unsigned> value = 0;
    std::thread t([&]() {
        for (size_t i = 0; i < 10; ++i)
            value.fetch_add(1); // OK, atomic increment
    });

    for (size_t i = 0; i < 10; ++i)
        value.fetch_add(1); // OK, atomic increment

    t.join();

    // value will contain 20
}
```

Semantics of Atomic Operations

C++ may support atomic operations that are not supported by the CPU

- `std::atomic<T>` can be used with any trivially copyable type
- In particular also for types that are much larger than one cache line
- To guarantee atomicity, compilers are allowed to fall back to mutexes

The C++ standard defines precise semantics for atomic operations

- Every atomic object has a totally ordered *modification order*
- There are several *memory orders* that define how operations on different atomic objects may be reordered
- The C++ memory orders do not necessarily map precisely to memory orders defined by a CPU

Modification Order (1)

All modifications of a *single* atomic object are totally ordered

- This is called the *modification order* of the object
- All threads are guaranteed to observe modifications of the object in this order

Modifications of *different* atomic objects may be unordered

- Different threads may observe modifications of multiple atomic objects in a different order
- The details depend on the *memory order* that is used for the atomic operations

Modification Order (2)

Example

```
std::atomic<int> i = 0, j = 0;
void workerThread() {
    i.fetch_add(1); // (A)
    i.fetch_sub(1); // (B)
    j.fetch_add(1); // (C)
}
void readerThread() {
    int iLocal = i.load(), jLocal = j.load();
    assert(iLocal != -1); // always true
}
```

Observations

- Reader threads will never see a modification order with `(B)` before `(A)`
- Depending on the memory order, multiple reader threads may see any of `(A),(B),(C)`, or `(A),(C),(B)`, or `(C),(A),(B)`

Memory Order (1)

The atomics library defines several memory orders

- All atomic functions take a memory order as their last parameter
- The two most important memory orders are `std::memory_order_relaxed` and `std::memory_order_seq_cst`
- `std::memory_order_seq_cst` is used by default if no memory order is explicitly supplied
- You should stick to this default unless you identified the atomic operation to be a performance bottleneck

```
std::atomic<int> i = 0;

i.fetch_add(1); // uses std::memory_order_seq_cst
i.fetch_add(1, std::memory_order_seq_cst);
i.fetch_add(1, std::memory_order_relaxed);
```

Memory Order (2)

`std::memory_order_relaxed`

- Roughly maps to a CPU with weak memory order
- Only consistent modification order is guaranteed
- Atomic operations of different objects may be reordered arbitrarily

```
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_relaxed); // (A)
        i.fetch_sub(1, std::memory_order_relaxed); // (B)
        j.fetch_add(1, std::memory_order_relaxed); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

Observations

- `threadB()` may observe `(A),(B),(C)`
- `threadC()` may observe `(C),(A),(B)`

Memory Order (3)

`std::memory_order_seq_cst`

- Roughly maps to a CPU with strong memory order
- Guarantees that all threads see all atomic operations in one globally consistent order

```
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_seq_cst); // (A)
        i.fetch_sub(1, std::memory_order_seq_cst); // (B)
        j.fetch_add(1, std::memory_order_seq_cst); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

Observations

- `threadB()` may observe `(C),(A),(B)`
- `threadC()` will then also observe `(C),(A),(B)`

Compare-And-Swap Operations (1)

Compare-and-swap operations are one of the most useful operations on atomics

- Signature: `bool compare_exchange_weak(T& expected, T desired)`
- Compares the current value of the atomic to `expected`
- Replaces the current value by `desired` if the atomic contained the expected value and returns `true`
- Updates expected to contain the current value of the atomic object and returns `false` otherwise

Often the main building block to synchronize data structures without mutexes

- Allows us to check that no modifications occurred to an atomic over some time period
- Can be used to implement “implicit” mutual exclusion
- Can suffer from subtle problems such as the A-B-A problem

Compare-And-Swap Operations (2)

Example: Insert into a lock-free singly linked list

```
#include <atomic>

class SafeList {
    private:
    struct Entry {
        T value;
        Entry* next;
    };

    std::atomic<Entry*> head;

    Entry* allocateEntry(const T& value);

    public:
    void insert(const T& value) {
        auto* entry = allocateEntry(value);
        auto* currentHead = head.load();
        do {
            entry->next = currentHead;
        } while (!head.compare_exchange_weak(currentHead, entry));
    }
};
```

Compare-And-Swap Operations (3)

`std::atomic` actually provides two CAS versions with the same signature

- `compare_exchange_weak` – weak CAS
- `compare_exchange_strong` – strong CAS

Semantics

- The weak version is allowed to return false, even when no other thread modified the value
- This is called "spurious failure"
- The strong version may use a loop internally to avoid this
- General rule: If you use a CAS operation in a loop, always use the weak version

std::atomic_ref (1)

std::atomic can be unwieldy

- std::atomic is neither movable nor copyable
- As a consequence it cannot easily be used in standard library containers

std::atomic_ref allows us to apply atomic operations to non-atomic objects

- The constructor takes a reference to an arbitrary object of type T
- The referenced object is treated as an atomic object during the lifetime of the std::atomic_ref
- std::atomic_ref defines similar member functions to std::atomic

Data races between accesses through std::atomic_ref and non-atomic accesses are still undefined behavior!

std::atomic_ref (2)

Example

```
#include <atomic>
#include <thread>
#include <vector>

int main() {
    std::vector<int> localCounters(4);
    std::vector<std::thread> threads;

    for (size_t i = 0; i < 16; ++i) {
        threads.emplace_back([&]() {
            for (size_t j = 0; j < 100; ++j) {
                std::atomic_ref ref(localCounters[i % 4]);
                ref.fetch_add(1);
            }
        });
    }

    for (auto& thread : threads) {
        thread.join();
    }
}
```

Organizing Larger Projects

Overview

Up to now a project scaffold has (mostly) been provided to you

- A substantial challenge in larger projects is simply organizing the project itself
- Bad project organization incurs enormous unnecessary overhead, promotes bugs, impedes extensibility and maintainability, ...

This lecture attempts to give some suggestions and an overview of useful tools

- Project layout suggestions (tailored to CMake)
- Integrating third-party tools and libraries with CMake
- Advanced debugging facilities
- We do not claim completeness or bias-free presentation
- Refer to the CMake documentation for much more detail

Project Layout (1)

The general project layout affects several interconnected properties

- Directory and source tree structure
- Namespace structure
- Library and executable structure

Changes to one of these properties likely entail changes to the other properties

- Namespace structure should (roughly) reflect directory structure and vice-versa
- Different libraries and executables ideally reside in separate source trees (i.e. directories)

Project Layout (2)

The project layout will evolve as a project grows

- Different guidelines apply to projects of different size
- Things one might get away with in small projects can become major issues in large projects
- Things that might be necessary in large projects can be overkill in small projects
- If a project is known to grow to a large size it pays off to plan ahead
- Definition of “small” and “large” is subjective

General guidelines

- *Always* clearly organize files, directories and namespaces with modularization in mind
- Start with a monolithic library/executable structure and move to a more independent and modular structure as the project grows

Directory Structure

General directory structure guidelines

- Files belonging to different libraries and executables should reside in different directories
- Files belonging to different components (logically separate parts) within a library or executable should reside in different directories
- Files belonging to different top-level namespaces should reside in different directories
- Tests should reside in a separate directory tree from the actual implementation
- Out-of-source builds should *always* be preferred

Directory Structure: Small Projects (1)

Directory structure guidelines for small projects

- The general directory structure guidelines still apply
- Parts of the `CMakeLists.txt` may be shared by all components within the project
  - Build system setup (e.g. compiler flags)
  - Dependencies (e.g. third-party libraries)
- The test code and executable(s) may be shared by all components within the project

Evolution

- Eventually, some library or executable in a small project will grow large
- Should then be moved into an independent (sub-)project

Directory Structure: Small Projects (2)

Small project example

```
> tree project
project
├── CMakeLists.txt          # Common CMakeLists.txt logic
├── my_executable
│   ├── CMakeLists.txt      # CMakeLists.txt logic for my_executable
│   └── ...                 # Source (& header) files
├── my_library
│   ├── CMakeLists.txt      # CMakeLists.txt logic for my_library
│   └── ...                 # Source (& header) files
└── test
    ├── CMakeLists.txt      # CMakeLists.txt logic for testing
    ├── my_executable
    │   └── ...             # Tests for my_executable
    └── my_library
        └── ...             # Tests for my_library
```

Directory Structure: Large Projects (1)

Directory structure guidelines for large projects

- The general directory structure guidelines still apply
- The components of large projects should be mostly independent subprojects
- Should *not* share most `CMakeLists.txt` logic
- Should *not* share test code and executable(s)

Evolution

- Eventually other projects or people may want to reuse one of the subprojects in a different context
- Should then be moved into an entirely independent project

Directory Structure: Large Projects (2)

Large project example

```
> tree project
project
├── CMakeLists.txt          # Minimal common CMakeLists.txt logic
├── my_executable
│   ├── CMakeLists.txt      # Common my_executable CMakeLists.txt
│   ├── src
│   │   └── ...             # Source (& header) files
│   └── test
│       ├── CMakeLists.txt  # CMakeLists.txt logic for tests
│       └── ...             # Tests for my_executable
└── my_library
    ├── CMakeLists.txt      # Common my_library CMakeLists.txt
    ├── src
    │   └── ...             # Source (& header) files
    └── test
        ├── CMakeLists.txt  # CMakeLists.txt logic for tests
        └── ...             # Tests for my_library
```

Header and Implementation Files

File content

- Generally, there should be one separate pair of header and implementation files for each C++ class
- Very tightly coupled classes (e.g. classes that could also be nested classes) can be placed in the same header and implementation files

File location

- Option 1: Place associated implementation and header files in the same directory (preferred by us)
- Option 2: Place associated implementation and header files in separate directory trees (e.g. `src` and `include`)
- Option 1 makes browsing code somewhat easier, option 2 makes system-wide installation easier

Namespaces & Cycles

Namespaces should identify logically coherent components within a library or executable

- Usually, there should be at least a top-level namespace (i.e. don't put stuff in the default namespace)
- Namespaces should group broadly similar or coherent functionality
- Rule of thumb: Think of namespaces as "candidates for moving into a separate library"

Dependencies between namespaces should be cycle-free

- Makes refactoring code *much* easier
- Allows future modularization into separate libraries

Library & Executable Structure

It is usually advisable to separate executables from their core functionality

- Executables often serve as "frontends" to some library functionality
- Library functionality can probably be reused in other programs
- Keeps interaction logic (e.g. I/O) separate from core functionality
- Not necessary in very small projects

There should be a separate `CMakeLists.txt` for each library or executable

- Implies that separate libraries and executables reside in separate directories
- Facilitates future modularization into separate (sub-)projects
- The `add_subdirectory` CMake function can be used to aggregate several such sub-projects

Include Directories

Usually, the include path for a library should contain a prefix

- E.g. includes for a library “foo” could start with `#include "foo/..."`
- Requires a suitable directory structure in the source tree of the library
- Usually requires the use of `target_include_directories` in the `CMakeLists.txt`

```
> tree project/my_library
my_library
├── CMakeLists.txt
├── src
│   └── my_library
│       ├── Bar.hpp
│       ├── Foo.hpp
│       └── Foo.cpp
└── test
    └── ...
```

Libraries & Executables

In most cases, libraries and executables are the main product of a CMake project

- Encoded as *targets* in a CMake project
- Targets can have properties such as dependencies
- CMake projects may contain further targets (e.g. for installing, packaging, linting, etc.)

Libraries

- Collection of compiled code that can be reused in other libraries or executables
- Can either be *static* or *shared* libraries
- Have to conform to the OS application binary interface (ABI)
- Cannot be executed on their own

Executables

- Compiled code that can be executed on a certain operating system
- Have to conform to the OS application binary interface (ABI)
- May contain further metadata such as information about entry points etc.

Executables in CMake (1)

Executables are added with the `add_executable` CMake command

- Syntax: `add_executable(name sources...)`
- Adds a CMake target with the specified name
- Produces an executable with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Executables in CMake (2)

Sample `CMakeLists.txt` for the `my_executable` sub-project

```
set(MY_EXECUTABLE_SOURCES
    src/my_executable/Helper.cpp
    ...
    src/my_executable/Main.cpp
)

add_executable(my_executable ${MY_EXECUTABLE_SOURCES})

# further commands required
```

Static Libraries

Static libraries are essentially archives of executable code

- Contain assembly from some number of object files, e.g. for classes, functions, etc.
- Dependencies on static libraries are resolved at *link time*
- Static libraries on Linux typically have the extension `*.a`

The linker is responsible for resolving dependencies on static libraries

- Code from a static library `A` is copied into a library or executable `B` that depends on `A`
- At runtime, no dependency on `A` exists since the relevant code is part of the library or executable `B`

Shared Libraries

Shared libraries are dynamic archives of executable code

- Contain assembly from some number of object files, e.g. for classes, functions, etc.
- Dependencies on shared libraries are resolved at *runtime*
- Shared libraries on Linux typically have the extension `*.so`

The operating system is responsible for resolving dependencies on shared libraries

- Only pointers to the code in a shared library `A` are used in a library or executable `B` that depends on `A`
- At runtime, the operating system loads `A` into memory once
- All programs depending on `A` access this memory to execute code in `A`

Advantages and Disadvantages of Static Libraries

Advantages

- Can have slightly higher performance since there are no indirections
- Can prevent compatibility issues since there are no external dependencies

Disadvantages

- Much bigger file sizes than shared libraries since code is actually copied
- Programs depending on static libraries have to be recompiled if the static library changes
- Can lead to problems with transitive dependencies even if they are "header only"

Advantages and Disadvantages of Shared Libraries

Advantages

- Much smaller file sizes since the shared library is only loaded into memory at run time
- Much lower memory consumption since only a single copy of a shared library is kept in memory (even for unrelated processes)
- Can be exchanged for other compatible versions without changing programs that depend on a shared library

Disadvantages

- Programs depending on a shared library rely on a compatible version being available
- Can be slightly slower due to additional indirection at runtime

Static Libraries in CMake (1)

Static libraries are added with the `add_library` CMake command

- Syntax: `add_library(name STATIC sources...)`
- Adds a CMake target with the specified name
- Produces a static library with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Static Libraries in CMake (2)

Sample `CMakeLists.txt` for the `my_library` sub-project (assuming static library)

```
set(MY_LIBRARY_SOURCES
    src/my_library/ClassA.cpp
    ...
    src/my_library/ClassZ.cpp
)

add_library(my_library STATIC ${MY_LIBRARY_SOURCES})

# further commands required
```

Shared Libraries in CMake (1)

Shared libraries are added with the `add_library` CMake command

- Syntax: `add_library(name SHARED sources...)`
- Adds a CMake target with the specified name
- Produces a shared library with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Shared Libraries in CMake (2)

Sample `CMakeLists.txt` for the `my_library` sub-project (assuming shared library)

```
set(MY_LIBRARY_SOURCES
    src/my_library/ClassA.cpp
    ...
    src/my_library/ClassZ.cpp
)

add_library(my_library SHARED ${MY_LIBRARY_SOURCES})

# further commands required
```

Interface Libraries in CMake (1)

Usually *only* the implementation files (`*.cpp`) should be added to a CMake target

- Header files on their own are not compiled
- Only headers that are included by implementation files are relevant for compilation

Exception: Interface libraries

- Syntax: `add_library(name INTERFACE)`
- A library might contain only template definitions
- Cannot be compiled into a static or shared library (unless explicit instantiation is used)
- Can still have properties such as include paths or dependencies

Interface Libraries in CMake (2)

Sample `CMakeLists.txt` for the `my_library` sub-project (assuming header-only)

```
add_library(my_library INTERFACE)
target_include_directories(my_library INTERFACE src)
target_link_libraries(my_library INTERFACE some_dependency)
```

Nested Projects in CMake (1)

The `add_subdirectory` CMake command can be used to add a subproject

- Syntax: `add_subdirectory(source_dir)`
- Adds the `CMakeLists.txt` in the specified `source_dir` to the build
- The nested `CMakeLists.txt` will be processed immediately by CMake
- The CMake variable `CMAKE_SOURCE_DIR` refers to the top-level source directory inside nested `CMakeLists.txt`
- The CMake variable `CMAKE_CURRENT_SOURCE_DIR` refers to the source directory in which the nested `CMakeLists.txt` resides

Nested Projects in CMake (2)

Example top-level `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.12)
project(project)

# more general setup code ...

add_subdirectory(my_executable)
add_subdirectory(my_library)
```

Important Project Properties (1)

Usually, the include directory of libraries and executables needs to be set

- `target_include_directories(target PUBLIC|PRIVATE dirs...)`
- Should be set to the `src` or `include` directory of a subproject in our suggested layout
- `PUBLIC` include directories are passed on to targets that depend on the current target

Important Project Properties (2)

Dependencies between targets can be set with `target_link_libraries`

- `target_link_libraries(target PUBLIC|PRIVATE libs...)`
- `libs...` can refer to libraries defined by the current project or imported third-party library targets
- `PUBLIC` dependencies are passed on to targets that depend on the current target

Important Project Properties (3)

Sample CMakeLists.txt for the my_executable sub-project

```
set(MY_EXECUTABLE_SOURCES
    src/my_executable/Helper.cpp
    ...
    src/my_executable/Main.cpp
)

add_executable(my_executable ${MY_EXECUTABLE_SOURCES})
# allows includes to be '#include "my_executable/..."
# instead of '#include "my_executable/src/my_executable/..."
target_include_directories(my_executable PRIVATE src/)
# dependency on the my_libary target defined in other subproject
target_link_libraries(my_executable PRIVATE my_library)
```

Paths in CMake

CMake defines several variables for often-used paths

`CMAKE_SOURCE_DIR`
: Contains the full path to the top level of the source tree, i.e. the location of the top-level `CMakeLists.txt`

`CMAKE_CURRENT_SOURCE_DIR`
: Contains the full path the the source directory that is currently being processed by CMake. Differs from `CMAKE_SOURCE_DIR` in directories added through `add_subdirectory`.

`CMAKE_BINARY_DIR`
: Contains the full path to the top level of the build tree, i.e. the build directory in which `cmake` is invoked.

`CMAKE_CURRENT_BINARY_DIR`
: Contains the full path the binary directory that is currently being processed. Each directory added through `add_subdirectory` will create a corresponding binary directory in the build tree.

Relative paths are usually relative to the current source directory

Third-Party Libraries

Usually we do not want to reinvent the wheel

- There is a vast ecosystem of (open-source) third-party libraries
- If there exists a well-maintained third-party library that matches your requirements you should use it

If possible and feasible, your project should *not* bundle third-party dependencies

- Many libraries can easily be installed through a package manager
- Reduces complexity of project configuration and maintenance
- CMake provides facilities for locating third-party dependencies in a platform-independent way

find_package (1)

Preferred CMake function for locating third-party dependencies

- `find_package(<PackageName> [version] [REQUIRED])`
- Finds and loads settings from an external project
- Sets the `<PackageName>_FOUND` CMake variable if the package was found
- May provide additional variables and imported CMake targets depending on the package

`find_package` relies on CMake scripts

- Attempts to find a `Find<PackageName>.cmake` file in the path specified by the `CMAKE_MODULE_PATH` variable and in the CMake installation
- Many `Find*.cmake` scripts are provided by CMake itself
- CMake documentation can be consulted for details about provided `Find*.cmake` scripts
- Own `Find*.cmake` scripts can be written if necessary

find_package (2)

Example

```
...

# Attempt to locate system-wide installation of libgtest
# Invokes the FindGTest.cmake script provided by CMake
# Configuration will fail if libgtest cannot be found
find_package(GTest REQUIRED)

add_executable(tester ...)
target_link_libraries(tester PRIVATE
    ...
    GTest::GTest  # Imported target for the gtest library
                  # as specified by the documentation of
                  # FindGTest
)
```

find_library (1)

If no Find*.cmake script is available, find_library can be used

- `find_library(<VAR> name [path1 path2 ...])`
- Creates a cache entry named <VAR> to store the result of the command
- If nothing is found, the result will be <VAR>-NOTFOUND
- name specifies the name of the library (e.g. gtest for libgtest)
- Additional paths beside the default search paths can be specified

find_library simply searches directories for a library

- A wide range of (highly configurable) paths is searched for the library
- Does not automatically configure non-standard include paths like find_package
- Should only be used as a fallback or within Find*.cmake scripts

find_library (2)

Example (assuming there is no `FindGTest.cmake` script)

```
...

# Attempt to locate libgtest library
# Searches for the library file in a range of paths
find_library(GTest gtest)

if (${GTest} STREQUAL "GTest-NOTFOUND")
    message(FATAL_ERROR "libgtest not found")
endif()

add_executable(tester ...)
target_link_libraries(tester PRIVATE
    ...
    GTest  # Only adds the libgtest library
           # Does not set include paths
)
```

Further Reading

We only scratched the surface of CMake in this lecture

- CMake provides much more highly useful functionality
- E.g. checks for compiler flags
- E.g. checks for compiler features
- E.g. checks for host system features
- E.g. defining custom Makefile targets
- ...

The CMake documentation provides a good overview

Testing

Tests should be an integral part of every larger project

- Unit tests
- Integration tests
- ...

Good test coverage greatly facilitates implementing a large project

- Tests can ensure (to some extent) that modifications do not break existing functionality
- Can easily refactor code
- Can easily change the internals of a component
- ...

Googletest (1)

We use Googletest in the programming assignments and final project

- Works on a large variety of platforms
- Contains a large set of useful functions
- Can usually be installed through a package manager
- Can be added to a CMake project through the `FindGTest.cmake` module
- Alternative test frameworks are of course available

Functionality overview

- Test cases
- Predefined and user-defined assertions
- Death tests
- ...

Googletest (2)

Simple tests

```
#include <gtest/gtest.h>
//-----------------------------------------------------------------------
TEST(TestSuiteName, TestName) {
    ...
}
```

- Defines and names a test function that belongs to a test suite
- Test suites can for example map to one class or function
- Googletest assertions can be used to control the outcome of the test function
- If any assertion fails or the test function crashes, the entire test case fails

Googletest (3)

Fatal assertions

- Fatal assertions are prefixed with `ASSERT_`
- When a fatal assertion fails the test function is immediately terminated

Non-fatal assertions

- Non-fatal assertions are prefixed with `EXPECT_`
- When a non-fatal assertion fails the test function is allowed to continue
- Nevertheless the test case will fail
- All assertions exist in fatal and non-fatal versions

Assertion examples

- `ASSERT_TRUE(condition);` or `ASSERT_FALSE(condition);`
- `ASSERT_EQ(val1, val2);` or `ASSERT_NE(val1, val2);`
- ...

Googletest (4)

A custom `main` function needs to be provided for Googletest

```
#include <gtest/gtest.h>
//----------------------------------------------------------
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- Should usually be placed in a separate `Tester.cpp` or `main.cpp`

Coverage (1)

Code coverage can help ensure proper testing of a project

- Simple metrics like line coverage have to be interpreted carefully
- Can indicate that a certain part of a project has *not* been tested properly
- Can usually *not* indicate that a certain part of a project has been tested exhaustively

Line coverage information can automatically be collected during test execution

- Possible with a variety of tools
- GCC contains the build-in coverage tool `gcov`
- Clang can produce `gcov`-like output
- `lcov` together with `genhtml` can be used to generate HTML line coverage reports from information collected during test execution

Coverage (2)

Brief example

```
# build executable with gcov enabled
> g++ -fprofile-arcs -ftest-coverage -o main main.cpp

# run executable and generate coverage data
> ./main

# generate lcov report
> lcov --coverage --directory . --output-file coverage.info

# generate html report
> genhtml coverage.info --output-directory coverage
```

- Produces HTML coverage report in `coverage/index.html`
- Configuration for coverage reports should be part of CMake configuration

Continuous Integration

Platforms like GitLab provide *continuous integration* (CI) functionality

- Can automatically run tests or other checks each time some commits are pushed to GitLab
- Highly useful in larger projects with multiple contributors
- Can be used to enforce certain standards in a project (e.g. minimum line coverage, no failing tests etc.)
- Has to be taken seriously to be effective (e.g. refuse merge requests with failing CI tests etc.)

Configured through `.gitlab-ci.yml` file in the repository

- Rather complex initial server-side setup
- Already provided by our GitLab server
- `.gitlab-ci.yml` configures the CI for a certain GitLab repository
- Refer to the GitLab documentation for details

Linting

A *linter* performs static source code analysis

- Can detect some types of "bad" code
- Some forms of bugs
- Stylistic errors that may lead to bugs
- Suspicious constructs that may lead to bugs

`clang-tidy` is a `clang`-based C++ linter

- Widely available through package manager
- Highly configurable set of checks (e.g. through `.clang-tidy` file)
- Integrated in CLion
- Can be integrated in CMake configuration of a project

`perf` (1)

`perf` is a highly useful performance analysis tool for Linux

- Can profile any program using the standalone executable `perf`
- Can be integrated in a program by using the `perf` API
- Can interface with hardware and software performance counters

Standalone `perf` examples

- `perf stat [OPTIONS] command`
  - Run command and display information about event counts such as cache misses, branch misses etc.
- `perf record [OPTIONS] command`
  - Run command and sample a certain event on the instruction level
  - If possible, command should be built with debug symbols
- `perf report`
  - Analyze a file generated by `perf record`
  - Generates an interactive report that shows sampled event counts for each instruction.

perf (2)

perf stat example

```
> perf stat --detailed ./my_executable
...
Performance counter stats for './my_executable':

      56.505,78 msec task-clock                #    2,573 CPUs utilized
        854.187      context-switches          #    0,015 M/sec
          7.827      cpu-migrations            #    0,139 K/sec
        309.550      page-faults               #    0,005 M/sec
177.728.516.281      cycles                    #    3,145 GHz
 60.347.961.620      instructions              #    0,34  insn per cycle
 12.694.777.815      branches                  #  224,663 M/sec
     89.725.841      branch-misses             #    0,71% of all branches
 16.672.843.754      L1-dcache-loads           #  295,064 M/sec
  1.267.581.260      L1-dcache-load-misses     #    7,60% of all L1-dcache hits
    471.681.999      LLC-loads                 #    8,347 M/sec
    258.238.607      LLC-load-misses           #   54,75% of all LL-cache hits

21,964215591 seconds time elapsed

44,360970000 seconds user
16,626546000 seconds sys
```

Valgrind

Valgrind is a general-purpose dynamic analysis tool

- Mainly used for memory debugging, memory leak detection and profiling
- Essentially runs programs on a virtual machine, allowing tools to do arbitrary transformations on the program before execution
- Extremely high overhead compared to other tools like ASAN

Use cases

- Complex memory bugs that are not detected by simpler tools like the address sanitizer
- Complex profiling tasks

Reverse Debugging (1)

Regular debuggers like GDB can only step forward in the program

- Does not necessarily fit debugging requirements
- E.g. when a crash occurs, we would like to step *backwards* until we have found the source of the crash

Reverse debuggers provide such functionality

- Usually, a program run is recorded first
- Subsequently, the program run can be replayed reproducing the exact same behavior
- During debugging, execution can step forward and backward in time
- Example: `rr` by Mozilla

Reverse Debugging (2)

Buggy class

main.cpp

```
#include <cassert>
//---------------------------------------------------------
struct Foo {
    static constexpr int max = 15;
    int a = 0;

    void bar() {
        assert((a % 2) == 0);
        a = (a + 2) % max;
    }
};
//---------------------------------------------------------
int main() {
    Foo foo;
    for (unsigned i = 0; i < 16; ++i)
        foo.bar();
}
```

Reverse Debugging (3)

rr example

```
> g++ -g -o main main.cpp
> rr record main        # record execution of main, including crash
> rr replay             # start rr GDB session, will break at _start
...
(rr) continue           # continue program until crash
(rr) up 4               # go to Foo::bar stack frame
(rr) watch -l a         # hardware watchpoint for Foo::a
(rr) reverse-continue   # continue backwards, will break at SIGABRT
(rr) reverse-continue   # continue backwards, will break at watchpoint
Continuing.

Hardware watchpoint 1: -location a

Old value = 1
New value = 14
0x00005568dba67208 in Foo::bar (this=0x7fff75f38980) at main.cpp:9
9                         a = (a + 2) % max;
```

C++ Systems Programming on Linux

C++ Systems Programming on Linux

Until now, most topics were about *standard* C++. The standard does not contain everything that is useful for good systems programming, such as:

- Creating, removing, renaming files and directories
- Efficient reading and writing of files
- Direct manual memory allocation from the kernel
- Networking
- Management of processes and threads

The Linux kernel in particular has a very extensive user-space C-API that can be used to directly communicate with the kernel for all of those tasks.

POSIX and Linux API

POSIX is a standard that defines a C-API to communicate with the operating system.

- The POSIX API is supported by most Unix-like operating systems (e.g. Linux, Mac OS X)
- It is a pure C-API but can also be used directly in C++
- Consists of types, functions and constants defined in `<unistd.h>`, `<fcntl.h>`, various `<sys/*.h>` files, and more

Linux defines additional types, functions and constants for Linux-specific operations that are not defined by the standard.

- Documentation of the POSIX functions can be found in man pages (usually in section `3posix` or `3p`)
- Linux-specific functions are also documented in man pages (usually in section 2)

File Descriptors

A very central concept in the POSIX API are so called *file descriptors* (fds).

- File descriptors have the type `int`
- They are used as a "handle" to:
  - Files in the filesystem
  - Directories in the filesystem
  - Network sockets
  - Many other kernel objects
- Usually, fds are created by a function (e.g. `open()`) and must be closed by another function (e.g. `close()`)
- When working with fds in C++, the RAII pattern can be very useful

Opening and Creating Files (1)

To open and create files the `open()` function can be used. It must be included from `<sys/stat.h>` and `<fcntl.h>`.

- `int open(const char* path, int flags, mode_t mode)`
- Opens the file at path with the given `flags` and returns an fd for that file
- If an error occurs, `-1` is returned
- The third argument mode is optional and only required when a file is created
- `flags` is a bitmap (created with bitwise or) that must contain exactly one of the following flags:

`O_RDONLY`	Open the file only for reading.
`O_RDWR`	Open the file for reading and writing.
`O_WRONLY`	Open the file only for writing.

- `close()` must be used to close the fd returned by `open()` → RAII

Opening and Creating Files (2)

There are more flags that can be combined with bitwise or:

`O_CREAT`	If the file does not exist, it is created with the permission bits taken from the mode argument
`O_EXCL`	Can only be used in combination with `O_CREAT`. Causes `open()` to fail and return an error when the file exists.
`O_TRUNC`	If the file exists and it is opened for writing, *truncate* the file, i.e. remove all its contents and set its length to 0.

Example:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
    int fd = open("/tmp/testfile", O_WRONLY | O_CREAT, 0600);
    if (fd < 0) { /* error */ }
    else { close(fd); }
}
```

Reading and Writing from Files

To read from and write to files, `read()` and `write()` from the header `<unistd.h>` can be used.

- `ssize_t read(int fd, void* buf, size_t count)`
- `ssize_t write(int fd, const void* buf, size_t count)`
- `fd` must be a valid file descriptor
- `buf` must be a memory buffer which has a size of at least `count` bytes
- The return value indicates how many bytes were actually read or written (can be up to `count`)
- Both functions return `-1` when an error occurs
- Note: Both functions may wait until data can actually be read or written which can lead to deadlocks!

File Positions and Seeking (1)

For an opened file the kernel remembers the current position in the file.

- `read()` and `write()` start reading or writing from the current position
- They both advance the current position by the number of bytes read or written

The function `lseek()` (headers `<sys/types.h>` and `<unistd.h>`) can be used to get or set the current position.

- `off_t lseek(int fd, off_t offset, int whence)`
- `off_t` is a signed integer type
- The current position is changed according to `offset` and whence, which is one of the following:

`SEEK_SET`	The current position is set to `offset`
`SEEK_CUR`	`offset` is added to the current position
`SEEK_END`	The current position is set to the end of the file plus `offset`

- `lseek()` returns the value of the new position, or `-1` if an error occurred

File Positions and Seeking (2)

Example:

```
int fd = open("/etc/passwd", O_RDWR);
auto fileSize = lseek(fd, 0, SEEK_END);
lseek(fd, -4, SEEK_CUR);
write(fd, "test", 4); // overwrite the last 4 bytes
```

Note: The current position is shared between all threads. Generally, `read()`, `write()`, and `lseek()` should not be used concurrently on the same fd.

Reading and Writing at Specific Offsets

There also exist two functions that read or write from a file without using the current position: `pread()` and `pwrite()` from the header `<unistd.h>`.

- `ssize_t pread(int fd, void* buf, size_t count, off_t offset)`
- `ssize_t pwrite(int fd, const void* buf, size_t count, off_t offset)`
- Conceptually, those functions work like `lseek(fd, offset, SEEK_SET)` followed by `read()` or `write()`
- However, they do not modify the current position in the file
- Should be used when reading from and writing to files from multiple threads

Getting Metadata of Files

Meta data of files, such as the type of a file, its size, its owner, or the date it was last modified, can be read with `stat()` or `fstat()`. Required headers: `<sys/types.h>`, `<sys/stat.h>`, `<unistd.h>`.

- `int stat(const char* filename, struct stat* statbuf)`
- `int fstat(int fd, struct stat* statbuf)`
- The meta data of the file specified by `filename` or `fd` is written into `statbuf`
- Returns `0` on success, `-1` on error
- `struct stat` has several member variables:

`mode_t st_mode`	The file mode (`S_IFREG` for regular file, `S_IFDIR` for directory, `S_IFLNK` for symbolic link, …)
`uid_t st_uid`	The user id of the owner
`off_t st_size`	The total size in bytes
…	

Changing the Size of a File

Files can be resized by using the functions `truncate()` or `ftruncate()` from the headers `<sys/types.h>` and `<unistd.h>`.

- `int truncate(const char* path, off_t length)`
- `int ftruncate(int fd, off_t length)`
- Sets the size of the file specified by `path` or `fd` to `length` bytes
- If the new length is larger than the old, zero bytes are appended at the end
- Returns `0` on success, `-1` on error
- These functions are especially useful when files are used as a memory buffers, e.g. for a buffer manager of a database system

More File Functions

POSIX and Linux have many more functions that deal with files and directories:

`mkdir()`	Create a directory
`mkdirat()`	Create a subdirectory in a specific directory
`openat()`	Open a file in a specific directory
`unlink()`	Remove a file
`unlinkat()`	Remove a file from a specific directory
`rmdir()`	Remove an empty directory
`chmod()`/`fchmod()`	Change the permissions of a file
`chown()`/`fchown()`	Change the owner of a file
`fsync()`	Force changes to a file to be written
...	

Memory Mapping

POSIX defines the function `mmap()` in the header `<sys/mman.h>` which can be used to manage the virtual address space of a process.

- ```
  void* mmap(void* addr, size_t length, int prot, int flags,
             int fd, off_t offset)
  ```
- Arguments have different meaning depending on `flags`
- On error, the special value `MAP_FAILED` is returned
- Always: If a pointer is returned successfully, it must be freed with `munmap()`
- `int munmap(void* addr, size_t length)`
- `addr` must be a value returned from `mmap()`
- `length` must be the same value passed to `mmap()`
- RAII should be used to ensure that `munmap()` is called
  ```

Memory Mapping Files (1)

One use case for `mmap()` is to map the contents of a file into the virtual memory. To map a file, the arguments are used as follows:

- `addr`: hint for the kernel which address to use, should be `nullptr`
- `length`: length of the returned memory mapping (usually multiple of page size)
- `prot`: determines how the mapped pages may be accessed and is a combination (with bitwise or) of the following flags:

`PROT_EXEC`	pages may be executed
`PROT_READ`	pages may be read
`PROT_WRITE`	pages may be written
`PROT_NONE`	pages may not be accessed

- `flags`: should be either `MAP_SHARED` (changes to the mapped memory are written to the file) or `MAP_PRIVATE` (changes are not written to the file)
- `fd`: descriptor of an opened file
- `offset`: Offset into the file where the mapping should start (multiple of page size)

Memory Mapping Files (2)

Example of reading integers from file `/tmp/ints`:

```
int fd = open("/tmp/ints", O_RDONLY);
void* mappedFile = mmap(nullptr, 4096, PROT_READ, MAP_SHARED, fd, 0);
int* fileInts = static_cast<int*>(mappedFile);
for (int i = 0; i < 1024; ++i)
    std::cout << fileInts[i] << std::endl;
munmap(mappedFile, 4096);
close(fd);
```

- Note: This assumes that integers are written in binary format to the file!
- Using `mmap()` to read from large files is often faster than using `read()`
- This is because with `mmap()` data is directly read from and written to the file without copying it to a buffer first

Using mmap for Memory Allocation

`mmap()` can also be used to allocate memory by not associating it with a file.

- `flags` must be `MAP_PRIVATE | MAP_ANONYMOUS`
- `fd` must be `-1`
- `offset` must be `0`
- Other arguments have the same meaning
- Used by `malloc()` internally
- Should be used manually only to allocate very large regions of memory (at least several MBs)

Example of allocating 100 MiB of memory:

```
void* mem = mmap(nullptr, 100 * (1ull << 20),
                 PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,
                 -1, 0);
// [...]
munmap(mem, 100 * (1ull << 20));
```

Creating Processes with fork

The most common way to start a new process in Linux is using `fork()` from the headers `<sys/types.h>` and `<unistd.h>`.

- `pid_t fork()`
- When `fork()` is called, the process is duplicated (including its virtual memory with all memory mappings, open file descriptors, etc.)
- In the original process, `fork()` returns the process id of the new process, or `-1` if an error occurred
- In the new process, `fork()` returns `0`

```
std::cout << "start ";
if (fork() == 0) {
    std::cout << "new ";
} else {
    std::cout << "old ";
}
std::cout << "end ";
```

One possible output for this example is: `start old end new end`

Fine-Grained Process Creation with clone

For greater control over creating a process, `clone()` from `<sched.h>` (which is also used by `fork()` internally) should be used.

- ```
  int clone(int (*fn)(void*), void* child_stack,
            int flags, void* arg)
  ```
- Takes a function pointer that will be executed in the new process, the new stack pointer for the process, flags, and an argument that will be passed to the function
- Returns the process id of the new process
- `flags` is `0` or a bitwise or combination of the following:

`CLONE_FILES`	File descriptors are shared between old and new process
`CLONE_FS`	File system information is shared (e.g. the current directory)
`CLONE_VM`	Virtual memory is shared
`CLONE_PARENT`	The parent process of the new process will be the parent of the current process
`CLONE_THREAD`	The new process will be a thread in the same thread group

...
  ```

Executing Other Programs

To execute an entirely new program, `execve()` from `<unistd.h>` can be used.

- `int execve(const char* pathname, char* const argv[],`
  `char* const envp[])`
- `pathname` is the path to binary that should be executed
- `argv` is a pointer to a null-terminated array for the program arguments
- `envp` is a pointer to a null-terminated array for the environment variables
- On success, the new program is executed, so the function does not return
- On error, returns `-1`
- `execve()` replaces the virtual memory of the old program by the new, but it keeps all fds
- Is often used in combination with `fork()`

```
std::vector<const char*> args = {"/bin/ls", "/", nullptr};
std::vector<const char*> env = {"FOO=bar", nullptr};
if (fork() == 0) {
    execve("/bin/ls", args.data(), env.data());
}
```

Linux Threads and Processes

A process can consist of several threads. There exist several identifiers to distinguish processes:

TID: Unique identifier for each thread
PID: Identifier for processes. Equal for all threads within a process
TGID: Thread group identifier is a synonym for PID
PGID: Identifier for process groups. Equal for all processes within a process group (children, siblings, ...)

- The first process within a group will have the same value for all of the above.
- The thread with the TID equal to the PID is called leader of the thread group.
- Sometimes, programs display the TID and incorrectly call it PID.

Thread Pinning

Threads can control on which physical CPU cores they run by using `sched_setaffinity()` from <sched.h>.

- `int sched_setaffinitiy(pid_t pid, size_t cpusetsize`
  `const cpu_set_t* mask)`
- `pid` stands for the process id whose affinity should be set, or `0` which stands for the current thread
- `cpusetsize` must be set to `sizeof(cpu_set_t)`
- `mask` is a pointer to a `cpu_set_t` which describes which CPU cores the thread is allowed to run on
- Returns `0` on success, `-1` on error
- Variables of type `cpu_set_t` can be modified with `CPU_ZERO(cpu_set_t* set)` and `CPU_SET(int cpu, cpu_set_t* set)`

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(0, &set); CPU_SET(4, &set);
sched_setaffinity(0, sizeof(cpu_set_t), &set);
```

Signals

In POSIX systems like Linux, every process can receive *signals*.

- Signals can either be generated by hardware (e.g. on memory access violations) or by software (by using `kill()`)
- By default, a process is either terminated or does nothing when it receives a signal
- A process can set a *signal handler* function which will be called when a signal is received
- The most common signals are:

Signal	Default	Description
`SIGSEGV`	terminate	"segfault", invalid memory access
`SIGINT`	terminate	interrupt from user, usually by pressing Ctrl + C
`SIGTERM`	terminate	process is terminated
`SIGKILL`	terminate	process is killed (cannot be caught with a signal handler)
`SIGCHLD`	ignore	a child process terminated

Setting Signal Handlers (1)

Signal handlers can set by using `sigaction()` from the header `<signal.h>`.

- `int sigaction(int signum, const struct sigaction* act,`
  `struct sigaction* sigact)`
- `signum` is the signal whose signal handler should be changed
- `act` is a pointer to the signal handler that should be set, or `nullptr` if an existing signal handler should be removed
- If `sigact` is not `nullptr`, it will contain the old signal handler after the function returns
- Returns `0` on success, `-1` on error
- `struct sigaction` has several members, the most important one is: `void (*sa_handler)(int)`
- `sa_handler` is a function pointer that points to the signal handler function that takes the signal as only argument

Setting Signal Handlers (2)

As signal handlers can be called at any time while other code is running, they should avoid to interfere with memory that is currently accessed.

```
void handler(int /*signal*/) {
    std::cout << "Ctrl-C was pressed\n";
    std::exit(1);
}
struct sigaction s{}; // Use {} here to zero-initialize
s.sa_handler = handler;
sigaction(SIGINT, &s, nullptr);
```

Sending Signals

A process can send a signal to itself or other process by using `kill()` from the headers `<sys/types.h>` and `<signal.h>`.

- `int kill(pid_t pid, int sig)`
- `pid` is the process id of the process that should recieve the signal
- If `pid` is `0`, the signal is sent to all processes in the process group
- If `pid` is `-1`, the signal is sent to *all* processes for which the calling process has the permission
- Returns `0` on success, `-1` on error
- With the signals `SIGUSR1` and `SIGUSR2` (“user-defined signals”) this can be used for (limited) communication between processes

Inter-Process Communication with Pipes (1)

Using basic signals is often not sufficient for communication between processes. `pipe()` (from `<unistd.h>`) can be used instead which creates two fds that are connected to each other.

- `int pipe(int pipefd[2])`
- Takes a pointer to an array that can hold two integers
- Returns `0` on success, `-1` on error
- Creates a unidirectional connection between `pipefd[0]` and `pipefd[1]`
- Everything that is written to `pipefd[1]` can be read from `pipefd[0]`
- Both fds must be closed eventually

```
int fds[2];
pipe(fds);
int readfd = fds[0]; int writefd = fds[1];
write(writefd, "hello", 5);
char buffer[5];
read(readfd, buffer, 5); // buffer now contains "hello"
close(readfd); close(writefd);
```

Inter-Process Communication with Pipes (2)

`pipe()` is usually used in combination with `fork()`:

```
int fds[2]; pipe(fds);
int readfd = fds[0];
int writefd = fds[1];
if (fork() == 0) {
    // We only need to read from the parent, so close writefd
    close(writefd);
    char buffer[6]; buffer[5] = 0;
    read(readfd, buffer, 5);
    std::cout << "parent wrote: " << buffer;
    close(readfd);
} else {
    // Likewise, close readfd
    close(readfd);
    write(writefd, "hello", 5);
    close(writefd);
}
```

Error Handling

Most functions use `errno` from the header `<cerrno>` for error handling.

- `errno` is a global variable that contains an error code
- Is set when a function returns an error (e.g. by returning `-1`)
- All possible values for `errno` are available as constants:

`EINVAL`	Invalid argument
`ENOENT`	No such file or directory (e.g. in `open()`)
`EACCES`	Permission denied
`ENOMEM`	Not enough memory (e.g. for `mmap()`)
...	

- A description of the error can be retrieved with `std::strerror()` from `<cstring>`

Miscellaneous

Pointer Tagging on x86-64 (1)

Virtual addresses are translated to physical addresses by the MMU

- Virtual addresses are 64-bit integers on x86-64
- On x86-64, only the lower 48 bit of pointers are actually used
- The upper 16 bit of pointers are usually required to be zero

The upper 16 bit of each pointer can be used to store useful information

- Usually called *pointer tagging*
- Tagged pointers require careful treatment to avoid memory bugs
- If portability is desired, an implementation that works without pointer tagging has to be provided (e.g. through preprocessor defines)
- Allows us to modify two values (16 bit tag and 48 bit pointer) with a single atomic instruction

Pointer Tagging on x86-64 (2)

We can store different things in the upper 16 bit of pointers

- Up to 16 binary flags
- A single 16 bit integer
- ...

Guidelines

- Always wrap tagged pointers within a suitable data structure
- Do not expose tagged pointers in raw form
- Store tagged pointers as `uintptr_t` internally
- Use bit operations to access tag and pointer parts

Pointer Tagging on x86-64 (3)

Using the upper 16 bit to store information

```
static constexpr uint64_t shift = 48;
static constexpr uintptr_t mask = (1ull << shift) - 1;
//--------------------------------------------------------------------
uintptr_t tagPointer(void* ptr, uint64_t tag)
// Tag a pointer. Discards the upper 48 bit of tag.
{
    return (reinterpret_cast<uintptr_t>(ptr) & mask) | (tag << shift);
}
//--------------------------------------------------------------------
uint64_t getTag(uintptr_t taggedPtr)
// Get the tag stored in a tagged pointer
{
    return taggedPtr >> shift;
}
//--------------------------------------------------------------------
void* getPointer(uintptr_t taggedPtr)
// Get the pointer stored in a tagged pointer
{
    return reinterpret_cast<void*>(taggedPtr & mask);
}
```

Pointer Tagging on x86-64 (4)

Using the lower 16 bit to store information

```
static constexpr uint64_t shift = 16;
static constexpr uintptr_t mask = (1ull << shift) - 1;
//-------------------------------------------------------------------
uintptr_t tagPointer(void* ptr, uint64_t tag)
// Tag a pointer. Discards the upper 48 bit of tag.
{
    return (reinterpret_cast<uintptr_t>(ptr) << shift) | (tag & mask);
}
//-------------------------------------------------------------------
uint64_t getTag(uintptr_t taggedPtr)
// Get the tag stored in a tagged pointer
{
    return taggedPtr & mask;
}
//-------------------------------------------------------------------
void* getPointer(uintptr_t taggedPtr)
// Get the pointer stored in a tagged pointer
{
    return reinterpret_cast<void*>(taggedPtr >> shift);
}
```

Vectorization

Most modern CPUs contain vector units that can exploit data-level parallelism

- Apply the same operation (e.g. addition) to multiple data elements in a single instruction
- Can greatly improve the performance of suitable algorithms (e.g. image processing)
- Not all algorithms are amenable to vectorization

Overview

- Can be used through extensions to the x86 instruction set architecture
- Commonly referred to as single instruction, multiple data (SIMD) instructions
- Can be used in C/C++ code through *intrinsic* functions
- The Intel Intrinsics Guide provides an excellent documentation

SIMD Extensions

SIMD extensions have evolved substantially over time

- MMX
- SSE, SSE2, SSE3, SSE4
- AVX, FMA, AVX2, AVX-512

Modern CPUs retain backward compatibility with older instruction set extensions

- The CPU flags exposed in `/proc/cpuinfo` indicate which extensions are supported
- We will briefly introduce AVX (`avx` flag in `/proc/cpuinfo`)
- AVX should be supported on most reasonably modern CPUs

AVX Data Types

AVX data types and intrinsics are defined in the `<immintrin.h>` header

- AVX adds 16 registers which are 256 bits wide each
- Can hold multiple data elements
- Can be used through special opaque data types

AVX data types

- `__m256`: Can hold eight 32 bit floating point values
- `__m256d`: Can hold four 64 bit floating point values
- `__m256i`: Can hold thirty-two 8 bit, sixteen 16 bit, eight 32 bit or four 64 bit integer values
- Commonly referred to as *vectors* (not to be confused with `std::vector`)

Other SIMD extensions follow similar naming conventions for data types

AVX Intrinsics

Usually, there are separate intrinsics for each data type

- AVX intrinsics usually begin with `_mm256`
- Next is a name for the instruction (e.g. `loadu`)
- Finally, the data type is indicated
  - `ps` for `__m256`
  - `pd` for `__m256d`
  - `si256` for `__m256i`
- Example: `_mm256_loadu_ps`

We will only show intrinsics for `__m256` in the following

- Intrinsics for other data types usually follow similar patterns
- Exception: AVX does not contain many arithmetic operations on integer types (added in AVX2)

Constant Values

We cannot directly modify individual data elements in AVX data types

- We have to use intrinsics for that purpose
- Intrinsics usually return the result of a modification

We can create constant vectors

- `__m256 _mm256_set1_ps(float a)`
  - Returns a vector with all elements equal to a
- `__m256 _mm256_set_ps(float e7, ..., float e0)`
  - Returns a vector with the elements `e0, ..., e7`
- `__m256 _mm256_setr_ps(float e0, ..., float e7)`
  - Returns a vector with the elements `e0, ..., e7`

Loading and Storing

Loading data from memory

- `__m256 _mm256_load_ps(const float* addr)`
  - Load eight 32 bit floating point values from memory starting at `addr`
  - `addr` has to be aligned to a 32 byte boundary
- `__m256 _mm256_loadu_ps(const float* addr)`
  - Load eight 32 bit floating point values from memory starting at `addr`
  - `addr` does not have to be aligned beyond usual `float` alignment

Storing data to memory

- `void _mm256_store_ps(float* addr, __m256 a)`
  - Store eight 32 bit floating point values in `a` to memory starting at `addr`
  - `addr` has to be aligned to a 32 byte boundary
- `void _mm256_storeu_ps(float* addr, __m256 a)`
  - Store eight 32 bit floating point values in `a` to memory starting at `addr`
  - `addr` does not have to be aligned beyond usual `float` alignment

Arithmetic Operations

AVX provides many arithmetic operations on vectors

- All the usual arithmetic operations
- Bitwise operations on integer types
- ...

Example: Adding vectors

- `__m256 _mm256_add_ps(__m256 a, __m256 b)`
  - Adds the individual elements of the vectors `a` and `b`
  - Returns the result of the addition

Example

Computing the sum of elements in an `std::vector`

```
#include <immintrin.h>
#include <vector>
//----------------------------------------------------------------------
float fastSum(const std::vector<float>& vec) {
    __m256 vectorSum = _mm256_set1_ps(0);
    uint64_t index;
    for (index = 0; (index + 8) <= vec.size(); index += 8) {
        __m256 data = _mm256_loadu_ps(&vec[index]);
        vectorSum = _mm256_add_ps(vectorSum, data);
    }

    float sum = 0;
    float buffer[8];
    _mm256_storeu_ps(buffer, vectorSum);
    for (unsigned i = 0; i < 8; ++i)
        sum += buffer[i];
    for (; index < vec.size(); ++index)
        sum += vec[index];

    return sum;
}
```

Further Operations

AVX contains many more instructions

- Comparison operations on vectors
- Masked operations

Allows vectorization of many algorithms

- Vectorization is not guaranteed to improve performance
- Generally, compute-heavy algorithms benefit greatly from vectorization
- Algorithms with a lot of fine-grained branching or many loads and stores may not benefit
- Vectorization is always an *optimization* that should not be applied prematurely

Template Metaprogramming

Templates can be used for meta-programming at compile time.

- Template specializations can be used to select different types depending on template arguments
- Recursive templates can be used for basic “control flow”
- The standard library defines several useful templates in `<type_traits>`
- All types and values are generated at compile time, so can be used as constants or template parameters

Type Traits

Type traits can be used to analyze properties of arbitrary types:

```
constexpr bool a = std::is_arithmetic_v<int>; // true
constexpr bool b = std::is_class_v<int>; // false
constexpr bool c = std::is_class_v<std::vector<int>>; // true
constexpr bool d = std::is_move_assignable_v<std::vector<int>>; // true
```

They can also be used to generate new types:

```
using T1 = std::remove_reference_t<int&>; // T1 is int
using T2 = std::add_pointer_t<int>; // T2 is int*
// T3 is const std::vector<int>&
using T3 = std::add_const_t<std::add_lvalue_reference_t<std::vector<int>>>;
// my_uintptr_t is uint64_t on systems where the size of void* is 8 bytes,
// or uint32_t otherwise.
using my_uintptr_t =
    std::conditional_t<sizeof(void*) == 8, uint64_t, uint32_t>;
```

Using Type Traits

Using type traits can prevent code duplication. Common example: const and non-const versions of an iterator.

```
template <typename T>
class Container {
    private:
    template <bool isConst>
    class Iterator {
        public:
        using reference = std::conditional_t<isConst, const T&, T&>;
        // [...]
    };

    public:
    using iterator = Iterator<false>;
    using const_iterator = Iterator<true>;
};
```

The C++20 Standard

C++20 is the latest release of the C++ standard

- Adds some very cool features to the C++ standard
- We already covered many of the well-supported new features throughout this course (e.g. concepts)
- In the following we will give an overview of additional potentially very useful features

Compiler support for these features is improving although still intermittent

- Some features (e.g. modules) are not yet implemented completely by some compilers
- Some features (e.g. coroutines) may be implemented but affected by compiler bugs
- In any case: Use the latest compiler version available to you

Implementing Concepts (1)

We can use concepts to restrict the types used by templates instead of SFINAE.

```
#include <concepts>

template <typename T> requires std::floating_point<T>
T fdiv1(T a, T b) {
    return a / b;
}

template <typename T>
T fdiv2(T a, T b) requires std::floating_point<T> {
    return a / b;
}

template <std::floating_point T>
T fdiv3(T a, T b) {
    return a / b;
}
```

Implementing Concepts (2)

Concepts can be defined by a single bool (usually computed from type traits).

```
#include <type_traits>

template <typename T>
concept signed_concept =
    std::is_integral<T>::value && std::is_signed<T>::value;

template <typename T> requires signed_concept<T>
T foo(T a) {
    return a - 1;
}

int x = 0;
foo(x); // OK, returns -1
unsigned y = 0;
foo(y); // Compile time error: constraints not satisfied
```

Implementing Concepts (3)

Concepts can be defined by a requires clause using:

- Simple requirements: Require an expression to be valid
- Type requirements: Require a type to be valid
- Nested requirements: Require nested require clauses to be satisfied
- Compound requirements: Require properties on the return type of an expression

```
template <typename Allocator>
concept IsAllocator = requires(Allocator &a) {
    a.allocate(); // Simple requirement
    typename Allocator::value_type; // Type requirement
    // Nested requirements:
    requires std::default_initializable<Allocator>;
    requires std::movable<Allocator>;
    // Compound requirements:
    { a.allocate() } -> std::same_as<typename Allocator::value_type*>;
    { a.deallocate(std::declval<typename Allocator::value_type*>()) }
        -> std::same_as<void>;
};
```

Coroutines (1)

Regular function calls are strictly nested

- A function call suspends execution of the calling function, and resumes execution at the start of the called function
- Eventually, the called function returns and execution of the calling function resumes after the function call expression

Functions have state that has to be maintained across nested function calls

- Values of any local variables
- The instruction at which to resume execution after a function call
- Strict nesting of function calls allows for highly optimized state maintenance on the stack
- Strict nesting of function calls makes implementing asynchronous operations cumbersome

Coroutines (2)

Coroutines are functions that can be suspended and resumed (almost) arbitrarily

- Suspending a coroutine transfers execution back to the caller
- Resuming a suspended coroutine continues execution at the point it was suspended
- The state of a coroutine remains alive across suspensions (e.g. local variables)

Coroutines in C++ are implemented with the help of three new keywords

- `co_await <expr>`: Suspends the coroutine and returns control to the caller
- `co_yield <expr>`: Returns a value to the caller and suspends the coroutine
- `co_return <expr>`: Returns a value to the caller and finishes the coroutine

Coroutines (3)

Unfortunately, C++ coroutines are currently quite painful to use

- There is not yet any "coroutine standard library"
- In order to actually use any of the coroutine keywords, we have to implement a lot of (boilerplate) infrastructure ourselves
- The behavior of C++ coroutines is highly configurable through the details of this infrastructure implementation
- Overall, it is quite difficult to implement working coroutines

Further complications that will (hopefully) improve over time

- Compiler bugs in the implementation of coroutines
- Suboptimal compiler error messages for coroutines
- Suboptimal debugger support for coroutines

Modules (1)

Modules help structure large amounts of code into logical parts

- A module consists of multiple translation units called *module units*
- Module units can *import* other modules
- Module units can *export* certain declarations

Facilitates encapsulation of logically independent parts

- Exported declarations are visible to name lookup in translation units that import the module
- Other declarations are not visible to name lookup

Reduces compilation overhead

- Exported definitions are compiled into easy-to-parse binary format
- No need to recursively parse transitive includes

Modules (2)

Example

greeting.cpp

```
export module greeting;

import <string>;

export std::string getGreeting() {
    return "Hello world!";
}
```

main.cpp

```
import greeting;
import <iostream>;

int main() {
    std::cout << getGreeting() << std::endl;
}
```

Perfect Forwarding

You can forward parameters keeping the value type with `std::forward`

```
#include <iostream>
#include <utility>

void foo(int&& n) {
    std::cout << "rvalue overload, n=" << n << "\n";
}
void foo(int& n){
    std::cout << "lvalue overload, n=" << n << "\n";
}
template <typename T>
void dispatcher(T&& n){
    foo(std::forward<T>(n));
}
dispatcher(1); // rvalue overload, n=1
int i = 2;
dispatcher(i); // lvalue overload, n=2
```

Designated Initializers

C++20 introduces *designated initializers*

- Allows explicit initialization of class members by name
- This was already possible in C and supported by many compilers
- C++20 now supports a subset of what is allowed in C

```
struct Foo {
    int a;
    int b;
};
Foo f{ .a = 1, .b = 2 };
```

Bit Manipulation

The <bit> header introduces several functions for bit inspection and manipulation.

- `std::bit_cast`: Inspect the object representation (instead of using `reinterpret_cast` with potential undefined behavior)
- `std::endian`: Check the endianness of the system
- `std::has_single_bit`: Check if number is power of two
- `std::bit_ceil`, `std::bit_floor`: Find the next/previous power of two
- `std::rotl`, `std::rotr`: Rotate bits
- `std::countl_zero`: Count the number of consecutive zero bits starting from the most significant bit
- `std::popcount`: Count the number of one bits
- ...

Additional atomic types

C++20 introduces an atomic specialization of `std::shared_ptr`

```
#include <atomic>
#include <memory>

struct LargeObject {
    char data[1000];
};
std::atomic<std::shared_ptr<LargeObject>> object;

void readThreadSafe() {
    auto objectPtr = object.load();
    if (objectPtr)
        objectPtr->data; /* do something with objectPtr->data */
}

void replaceThreadSafe(std::shared_ptr<LargeObject> newObject) {
    object.store(std::move(newObject));
}
```

std::format

C++20 introduces a new way to format strings: `std::format`.

```
#include <format>

std::cout << std::format("Hello {}!", "world") << std::endl;
// Prints "Hello world!"
std::cout << std::format("{} {}!", "Hello", "world", "something")
          << std::endl;
// OK, prints "Hello world!"
```

Unfortunately, still limited or no support in many compilers.

More Features

C++20 introduces further small and large features, such as:

- `std::source_location`: Stores a location in the source code. `std::source_location::current()` can be used to get the location of the current line
- `<numbers>` header: Contains mathematical constants like `std::numbers::pi` and `std::numbers::e`
- `consteval` and `constinit`: Behave like a "mandatory" `constexpr`
- More functions and classes in the standard library are `constexpr`
- Some restrictions of lambdas were removed, e.g. you can capture structural bindings
- Non-type-template arguments can have a user-defined type
- ...

The C++23 Standard

C++23 will be the next release of the C++ standard

- Will add useful features such as a coroutine library, a modularized standard library, ...

Compiler support for new features is still experimental and very limited.

- Might be not be fully implemented in a while.
- In any case: Use the latest compiler version available to you