Concepts of C++ Programming

summer term 2022

PD Dr. Tobias Lasser

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Contents

Who?

- Lecture:
  - PD Dr. Tobias Lasser
    - Computational Imaging and Inverse Problems (CIIP)
- Exercises:
  - Alessandro Wollek, Jonas Jelten
    - Scientific staff at CIIP

What?

Aim of the lecture:

- study the concepts of modern C++ programming
- learn to apply those concepts in elegant and efficient solutions

Target audience:

- Bachelor students of Informatics (3rd term onwards)
- Master students of Informatics, BMC, CSE, Robotics
- anyone who is interested!

When and where?

Dates, places, and links

- Monday, 16:15 - 17:45, Galileo Hörsaal (8120.EG.001)
  - Live Stream, recording: `https://live.rbg.tum.de/`
  - Tweedback: `https://tweedback.de/`
- Wednesday, 16:15 - 17:45, Galileo Hörsaal (8120.EG.001)
  - Video conference: `https://bbb.rbg.tum.de/tob-w3o-4un-gp6`

Generally:

- Monday: lecture
- Wednesday: exercises

Tweedback Session-ID today: egy4

- URL: `https://tweedback.de/egy4`

Online resources

Chat platform

`https://zulip.in.tum.de`, Stream #CPP

- discussions, answering questions

Moodle course

`https://www.moodle.tum.de/course/view.php?id=75234`

- news and announcements
- materials (slides, exercise sheets etc.)

Website

`https://ciip.in.tum.de/teaching/cpp_ss22.html`

Contact and feedback

Questions

- chat: `https://zulip.in.tum.de`, Stream #CPP
- during lecture:
  - in person
  - Tweedback (today: session ID egy4)
- email: `lasser@in.tum.de` (in urgent cases only)

Feedback

- chat
- email

Feedback is always welcome!

Examination

- Written exam:
  - *date not set yet*
  - in presence
  - 90 minutes duration
  - no repeat exam!
- Homework:
  - weekly programming assignments
  - passing the automated tests yields 1 point per week (total of 12 points achievable, as currently planned)
- Final grade:
  - points from written exam and from homeworks added together
  - final grade will be derived from the sum of points

Homework assignments

Programming assignments

- each week you receive an exercise sheet (via Moodle) with some programming assignments
- submit your solutions via `https://gitlab.lrz.de`
- automatic tests run on your solutions

- you will get one point per homework assignment where you pass all the automated tests within the deadline (1 week)
- total achievable points: 12 (as currently planned)
- points will be added to the points achieved in the written exam to form the final grade

- details on how to submit the homework will be presented in the first exercise session (Wednesday, Apr. 27, 2022)

Homework assignments: brief details

- one homework assignment every Wednesday via Moodle
- deadline for automated tests to pass: +1 week (Wednesday 16:00)
- your solutions have to be uploaded to a GitLab repository

- how to get access to your GitLab repository:
  - log in to `https://gitlab.lrz.de`
  - edit your name (Profile, Account, Username) if you wish
  - in Moodle: go to "Add your GitLab username" activity
    - add your GitLab profile username (check `https://gitlab.lrz.de/-/profile/account` if unsure)
  - request access for the Waiting Room at `https://gitlab.lrz.de/cppcourse/ss2022/waiting-room`
  - your personal repository will be setup automatically

  - if there are any issues: use our chat at `https://zulip.in.tum.de`, stream #CPP

Lectures

Lecture format

- traditional slide-based lecture

- all materials will be available in Moodle
- feel free to ask questions any time via Tweedback!
  - I will try to make sure to pause often to answer your comments there

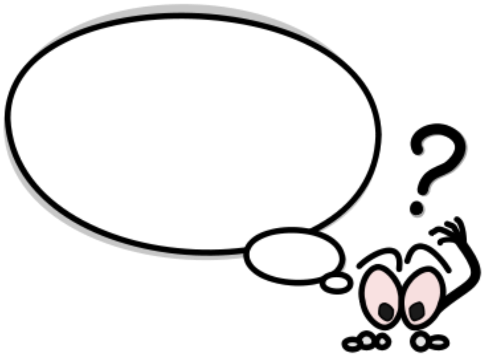
Exercises

Exercise format

- one exercise sheet per week
- exercise session will discuss
  - the previous week's assignments
  - give some pointers to the current exercise sheet

- all materials will be available in Moodle
- the level of interaction is up to you!

Questions?



Contents

Overview of topics

- basic concepts
  - syntax, strong typing, type deduction
  - compilation units, ODR
- concepts for build systems and dependency management
  - automatic compilation, linking, and dependencies
  - continuous testing and integration
- concepts for procedural programming
  - functions, parameter passing, lambdas, overloads, error handling
- concepts for object-oriented programming
  - classes, inheritance, polymorphism, RTTI
- concepts for generic programming
  - templates, variadic templates, fold expressions
  - CRTP and expression templates

Overview of topics (cont.)

- concepts for resource management
    - RAII, pointers, universal references, ownership, copy/move
- concepts for compile-time programming
    - constexpr, template recursion, SFINAE, type traits
- concepts for containers and iterators
    - containers in STL, iterators, algorithms, other STL types
- concepts for parallel programming
    - threads, atomics, async and futures, parallel STL
- outlook to future concepts (C++23 etc.)
    - coroutines, executors, networking, modular standard library

Please note

This course is still relatively new

$\rightarrow$ any feedback from you is welcome!

Survey

- Have you done anything in C++ yet?
- Have you done anything in C yet?

Contents

What is C++?

C++ is a multi-paradigm, general-purpose programming language, supporting:

- procedural programming
- object-oriented programming
- generic programming
- functional programming

Key characteristics:

- static typing
- compiled language
- facilities for low-level programming

Some C++ history

Early development:

- 1979: Bjarne Stroustrup begins work on C with Classes
- 1983: name changed to C++
- 1985: reference book The C++ Programming Language
  - also: release of the first commercial implementation

- 1998: standardization as ISO/IEC 14882:1998, C++98
- 2003: minor revision of ISO standard, C++03

Some C++ history (cont.)

Development of modern C++:

- 2011: new ISO standard (major revision), C++11
- 2014: minor revision of ISO standard, C++14
- 2017: revision of ISO standard, C++17
- 2020: revision of ISO standard, C++20
- planned for 2023: revision of ISO standard, C++23

Although modern C++ is (mostly) backwards compatible, it is almost a new programming language compared to C++98/03. The focus is on safe programming techniques with zero overhead.

Why C++?

- Performance:
  - high performance for all types (including user-defined)
  - low-level hardware access is possible
  - zero overhead rule: “You don’t pay for what you don’t use.” (B. Stroustrup)
- Flexibility:
  - flexible level of abstraction (very low-level to very high-level)
  - supports several programming paradigms
  - excellent scaling to very small and very large systems
  - interoperability with other programming languages
  - extensive ecosystem (tool chains, libraries)

Why C++? Examples

```cpp
template <class... Ts>
constexpr auto average (Ts... nums) requires (sizeof...(nums) > 0)
{
    return (nums + ...) / sizeof...(nums);
}
```

computes the average of basically anything

Why C++? Examples

```cpp
auto z = (log(c * exp(-1/2 * square(xn - mu) / sq))).sum();
```

this emits CUDA code at compile-time to compute Gaussian log likelihood (with the correct library)

Why C++? Examples

```cpp
for (auto n : std::views::iota(101, 200)
                 | std::views::filter([](auto v) { return v % 7 == 0; })
                 | std::views::take(3) )
    std::cout << n << "\n";
```

prints the last three numbers divisible by 7 in the range $[101, 200]$

Why C++? Examples

You can still do regular C:

```c
int main(int b,char**i){long long n=B,a=I^n,r=(a/b&a)>>4,y=atoi(*++i),
_=(((a^n/b)*(y>>T)|y>>S)&r)|(a^r);printf("%.8s\n",(char*)&_);}
```

(Source: winner of 2020 International Obfuscated C Code Contest)

Contents

A first C++ program

File first.cpp:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world!\n";
}
```

A first C++ program (cont.)

File first.cpp:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world!\n";
}
```

In order to run it, we have to compile it:

```
$ c++ -std=c++20 -Wall -Werror first.cpp -o first
```

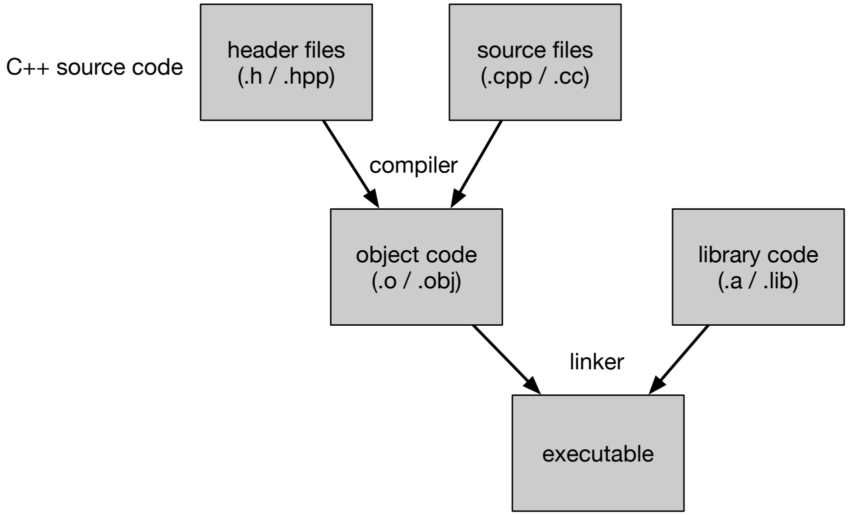
and then run it:

```
$ ./first
```

which produces:

```
Hello world!
```

Compilation and linking



Translation units

- the input of the compiler is called translation unit
- the output is the object code (.o / .obj)

- typically, a translation unit consists of:
  - a source file (.cpp / .cc)
  - some header files (.h / .hpp) that are `#include`'d

- `#include` is done via the preprocessor
  - in fact, the included file is literally included into the source code (which explains some of the long compilation times)
  - *side note:* this relic of C was finally superseded in C++20 (modules), but support of compilers / build systems is still lacking

Header and source files: example

sayhello.h:

```cpp
// declaration
void sayHello();
```

sayhello.cpp:

```cpp
#include <iostream>

#include "sayhello.h"

// definition
void sayHello() {
    std::cout << "Hello world!\n";
}

int main() {
    sayHello();
}
```

The purpose of the header/source split is to have

- declarations in header files
  - these are easy to include
  - also in other translation units
- definitions in source files

One definition rule

One definition rule (ODR)

- all objects (functions, variables, templates etc.) have *at most* one definition in any translation unit
- all objects or non-inline functions that are used have *exactly* one definition in the entire program
- some things like types, templates, and inline functions can be defined in more than one translation unit, but they must be effectively identical

For all subtleties and exceptions, please see the standard.

The job of compiler and linker

- the compiler uses the declaration to know the signature of the declared object (function, variable etc.)
- the compiler does **not** resolve the the symbols if they are defined externally (i.e. in another object file or library)

- the linker does resolve these symbols
- this often results in errors, for example:
  - missing symbols when forgetting to link an object file or library
  - multiple definitions of symbols when violating the ODR

ODR examples

In same translation unit, compiler catches ODR violations:

```cpp
int i{5}; // OK: declares and defines i
int i{6}; // ERROR: redefinition of i
```

Separate declaration and definition is required to break circular dependencies:

```cpp
    void bar(); // declares function bar
    void foo() { // declares and defines function foo
        bar();
    }

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

ODR examples (cont.)

File a.cpp:

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

File b.cpp:

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

int main() {}
```

Now the linker will catch the ODR violation:

```
$ c++ -c a.cpp -o a.o
$ c++ -c b.cpp -o b.o
$ c++ a.o b.o
duplicate symbol 'foo()' in:
    a.o
    b.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1
```

Header guards

- a file may transitively include the same header several times
- this can lead to violations of the ODR:

File headerI.h:

```cpp
int i = 1;
```

File headerJ.h:

```cpp
#include "headerI.h"

int j = i;
```

File main.cpp:

```cpp
#include "headerI.h"
#include "headerJ.h" // ERROR: i is defined twice
```

- solution: header files ensure that they are included at most once in a translation unit via header guards

Header guards: the portable way

File headerI.h:

```
// use any unique name
// usually composed from filename
#ifndef H_headerI
#define H_headerI

int i = 1;

#endif
```

File headerJ.h:

```
#ifndef H_headerJ
#define H_headerJ

#include "headerI.h"

int j = i;

#endif
```

Header guards: the elegant way

File headerI.h:

```
#pragma once

int i = 1;
```

File headerJ.h:

```
#pragma once

#include "headerI.h"

int j = i;
```

- drawback: this `#pragma once` is not in the C++ standard
- but all the big compilers support it

Outlook: C++20 modules

- C++20 modules replace the C-preprocessor based `#include` with proper modules using import/export
- example:

File sayhello.cpp:

```cpp
// declare and export module
export module sayHello;

// import standard io library
import <iostream>;

// export the function
export void sayHello() {
    std::cout << "Hello world!\n";
}
```

File main.cpp:

```cpp
// import our module
import sayHello;

int main() {
    sayHello();
}
```

Outlook: C++20 modules (cont.)

- modules also support the split of interface/implementation:

File sayhello.cppm:

```cpp
// declare and export module
export module sayHello;

// export the function
export void sayHello();
```

File main.cpp:

```cpp
// import our module
import sayHello;

int main() {
    sayHello();
}
```

File sayhello.cpp:

```cpp
// define module implementation
module sayHello;

// import standard io library
import <iostream>;

void sayHello() {
    std::cout << "Hello world!\n";
}
```

Contents

Getting a C++ compiler

Depending on your operating system, there are many options.

Here are some popular ones:

- Linux: get gcc or clang via your favorite package manager
- macOS:
  - installing Xcode will net you a clang version
  - alternatively, get gcc or clang via managers such as brew
- Windows:
  - get Visual Studio with Visual C++ (no cost for TUM students)
  - get Cygwin or MinGW with gcc
  - or use WSL to install gcc or clang

Getting a development environment

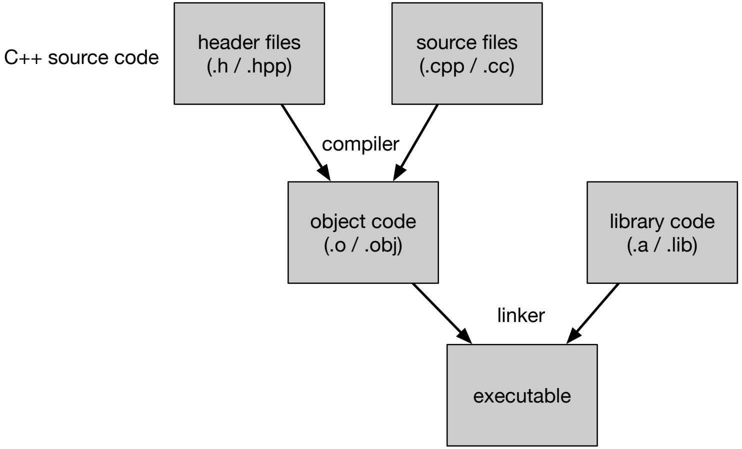
- any text editor (e.g. vim, Emacs) and a command line will do
- if you want more comfort, get an IDE, such as
  - QtCreator
  - Visual Studio Code
  - CLion (no cost for TUM students)

Investigate tools

- start getting familiar with important tools, such as
  - git for source code management
  - CMake for build automation and dependency management
  - a debugger (such as lldb or gdb) for debugging

Contents

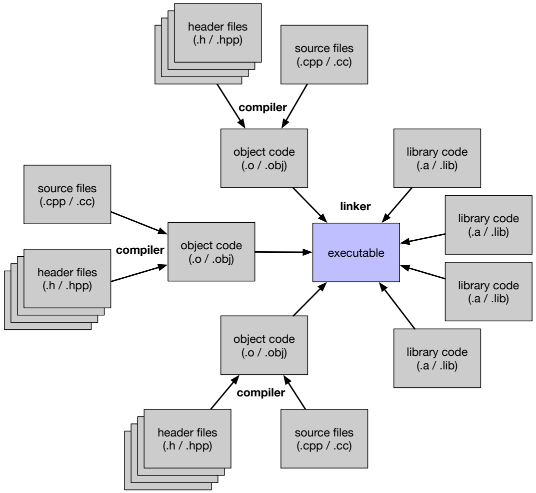
Compilation and linking



To compile and link:

```
$ c++ -std=c++20 -Wall -Wextra -O3 -c -o source.o source.cpp
$ c++ -std=c++20 -Wall -Wextra -O3 -o executable source.o library.a
```

More realistic example



To compile and link: many `$ c++ ...` calls

Automatic compilation and linking

- C++ projects usually consist of many source files (.cpp) and header files (.h) that need to be compiled and linked
    - tedious to do this manually
    - when one of the source files (.cpp) changes, only the corresponding code should be recompiled (not everything)
    - when one of the header files (.h) changes, only the source files that `#include` it should be recompiled

⇒ automate using a build system!

- examples are:
    - make, MSBuild, ninja, and many others
- we will use CMake, a build system that is also independent of the platform and the compiler

Contents

CMake

- CMake is a meta build system
    - independent of platform and compiler
    - specifies the build in CMakeLists.txt files
    - using its own language
    - generates output for other build systems, for example make files, MSBuild files, ninja files, and many others

- CMake is very commonly used for C++ projects
- it is natively supported by most IDEs (such as QtCreator, CLion, Visual Studio Code, or Visual Studio)

- documentation is available online, for example:
    - reference: `https://cmake.org/cmake/help/latest/`
    - guide: `https://cliutils.gitlab.io/modern-cmake/`

A basic CMakeLists.txt file

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.14)
project(exampleProject LANGUAGES CXX)

add_executable(exampleProject main.cpp)
```

Building:

```
$ mkdir build; cd build    # create separate build directory
$ cmake ..                 # generate make files
-- The CXX compiler identification is AppleClang 11.0.0.11000033
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/X/cmake_example/build
$ make                     # build the project
Scanning dependencies of target exampleProject
[ 50%] Building CXX object CMakeFiles/exampleProject.dir/main.cpp.o
[100%] Linking CXX executable exampleProject
[100%] Built target exampleProject
```

Important CMake commands

- `cmake_minimum_required(VERSION 3.14)`
  require (at least) a specific version (mandatory)
- `project(exampleProject LANGUAGES CXX)`
  define a C++ project with name "exampleProject"
- `add_executable(myProgram a.cpp b.cpp)`
  define an executable named "myProgram" (a target) built from the source files a.cpp and b.cpp
- `add_library(myLibrary c.cpp d.cpp)`
  define a library named "myLibrary" (a target)
- `target_include_directories(myProgram PUBLIC inc)`
  where to look for include files for the target "myProgram"
- `target_link_libraries(myProgram PUBLIC myLibrary)`
  which libraries to add to target "myProgram"

CMake variables

CMake uses variables to adapt/customize the build process.
Variables are set:

- in CMakeLists.txt: `set(FOO bar)`
  (set the variable `${FOO}` to “bar”)
- or on the command line: `cmake -D FOO=bar`
- or you can also use `ccmake` or `cmake-gui` to adjust variables manually

Important variables are:

- `CMAKE_CXX_COMPILER`
  set e.g. to “clang++” to use the clang compiler
- `CMAKE_BUILD_TYPE`
  set to “Debug” or “Release”, which affects compiler optimization and debug information

Target properties

Variables are global. It is preferred to use target-specific properties, such as:

- `target_include_directories(myProgram PUBLIC inc)`
  where to look for include files for “myProgram”
- `target_link_libraries(myProgram PUBLIC myLibrary)`
  which libraries to link to “myProgram”
- `target_compile_features(myProgram PUBLIC cxx_std_20)`
  enable the C++20 standard for “myProgram”

Properties can be PUBLIC or PRIVATE, determining whether those properties get propagated to other targets using this target (PUBLIC) or not (PRIVATE).

CMake and subdirectories

- larger projects are usually split up into subdirectories
- CMake expects the main CMakeLists.txt in the top-level folder to contain the `project()` command
  - the subdirectories are explicitly added using the command `add_subdirectory(subdir)`
  - the subdirectories contain their own CMakeLists.txt file (without `project()`)

CMake example directory structure

```
ExampleProject
├── CMakeLists.txt
├── myLibrary
│   ├── CMakeLists.txt
│   ├── myLibrary.h
│   ├── myLibrary.cpp
│   ├── awesomeFunctions.h
│   └── awesomeFunctions.cpp
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- the root CMakeLists.txt contains the `project()` statement
- it includes myLibrary using `add_subdirectory()`
- it includes src using `add_subdirectory()`

- for a bigger example, check out the homework assignments in GitLab!

Contents

Version control

- version control is essential in projects with many developers to be able to synchronize the code
- it has many advantages even when working alone
- git is currently the most popular version control system
  - open source
  - decentralized (i.e. no server required)
  - very efficient using branches and tags
- documentation available at:
  - reference: `https://git-scm.com/docs`
  - book: `https://git-scm.com/book/en/v2`
  - live explorer: `http://git-school.github.io/visualizing-git/`

Continuous integration

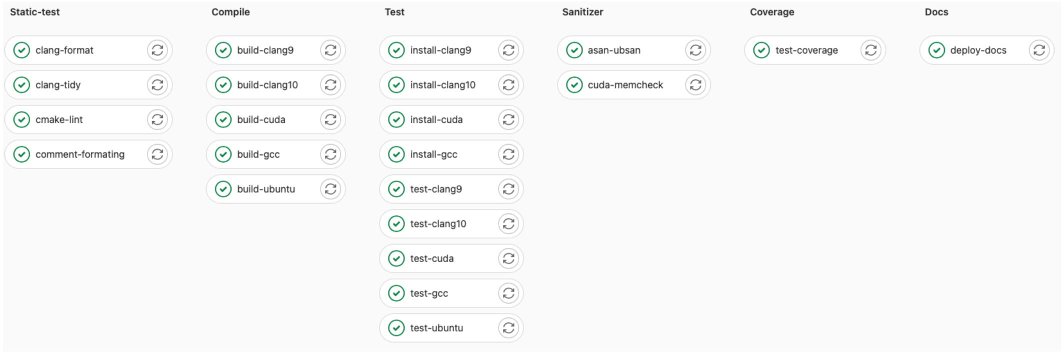
When you develop and push your changes to the git repository:

- you want to be sure that your code compiles successfully (ideally with different compilers)
- you want to be sure that all your tests pass

→ continuous integration runs automated builds/tests each time you push to the repository

- commonly used tools: Travis CI, Jenkins, or Gitlab runners
- steps often defined in a YAML file in the repository
- we use this (with Gitlab runners) for your homework!

CI: example



You can check this out in more detail yourself at `https://gitlab.lrz.de/IP/elsa`.

Contents

Finding help: the C++ reference

The basics of C++ are rather simple, just like almost any other programming language

- most of the complexity comes from many special cases and exceptions to many of the rules

We will not be covering every special case / exception!

- the lecture content might be inaccurate or incomplete at times
- you can find an accurate and complete reference documentation of C++ here:

`https://en.cppreference.com/w/cpp`

make sure to use it!

Minimal C++ program

The smallest C++ program:

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

- every C++ program must contain the special function `main()`
- program starts with `main()` (except for the exceptions...)
- program ends when `main()` ends (except for exceptions...)

The four elementary building blocks

Computable algorithms require four elementary building blocks:

- elementary processing step
- sequence of steps
- conditional processing step
- loop

The four elementary building blocks (cont.)

- elementary processing step:

```
int x{0};  // variable definition and initialization
```

- sequence of steps:

```
int x;     // sequence of steps marked by ';'
x = 0;
```

- conditional processing step:

```
if (x == 0)     // test condition in brackets (...)
    x = 2;      // executed if condition is true
else
    x = 0;      // executed if condition is false
```

- loop:

```
while (x > 0) { // execute body while condition true
    --x;        // body is a block enclosed in { }
}
```

Blocks (or scopes) in C++

- blocks are marked using curly brackets in C++:

```
if (x == 0)    // if there is only one statement
    x = 2;     // no block markings are necessary
else
    x = 0;
```

```
if (x == 0) { // but you can use blocks if you want
    x = 2;
}
else {        // you will need them if you want to
    x = 0;    // group several statements
    x -= 1;
}
```

- each block marks a scope (see variable lifetime later)

Loops

- variants of basic while loop:

```cpp
int a{6};
while (a > 0) {
    std::cout << a << "\n";
    --a;
}
```

```cpp
int a{6};
do {
    std::cout << a << "\n";
    --a;
} while (a > 0);
```

```cpp
for (int a{6}; a > 0; --a) {
    std::cout << a << "\n";
}
```

- these variants are functionally equivalent
  - however, in the for loop, the variable a is local to the for loop!

Loop control

- loops can be exited early using `break`
- can skip rest of one loop iteration using `continue`

```
int a{12};

while (a > 0) {
    --a;
    if (a % 2)    // skip rest of loop if a not divisible by 2
        continue;

    std::cout << a << "\n";

    if (a == 2)   // abort loop early if a == 2
        break;
}
```

Contents

Type safety

Ideal: a language has type safety

- meaning: every object will be used only according to its type
    - an object will be only used after initialization
    - only operations defined for the object's type will be applied
    - every operation leaves the object in a valid state

- static type safety
    - a program that violates type safety will not compile
    - the compiler (ideally) reports every violation
- dynamic type safety
    - if a program violates type safety it will be detected at run-time
    - a run-time system (ideally) detects every violation

Type safety (cont.)

C++ is neither statically nor dynamically type safe!

- it would interfere with being able to express ideas in code
- it would interfere with the performance goals

But: type safety is very important!

- try very hard not to violate it
- use the available tools:
  - most importantly, the compiler (and its warnings!)
  - static analysis tools (e.g. clang-tidy)
  - dynamic analysis tools (e.g. sanitizers)

C++ is a strongly typed language

- all objects in C++ require a type
- available types:
  - built-in types (e.g. int, float)
  - user-defined types (e.g. classes)

- declaration (and definition) anywhere within a scope (block)

```
TYPE variablename;
```

- initialization with curly brackets

```
TYPE variablename{initializer};
```

(more on this later)

Numbers

- built-in types to support:
  - natural numbers: unsigned using binary representation
  - whole numbers: signed using 2-complement representation (only required since C++20)
  - floating point numbers: using IEEE-754 standard

- all the regular operations are supported:
  - arithmetic: +, -, *, /, %
  - increment/decrement operators: ++, --
  - comparison: <, <=, >, >=, ==, !=
  - logical: && (and), || (or), ! (not)

Natural and whole numbers

Type	Equivalent type	guaranteed size	size in 32bit system	size in 64 bit system
short, short int signed short, signed short int	short int	>= 16 bit	16 bit	16 bit
unsigned short unsigned short int	unsigned short int			
int signed int, signed	int	>= 16 bit	32 bit	32 bit
unsigned unsigned int	unsigned int			
long, long int signed long, signed long int	long int	>= 32 bit	32 bit	64 bit (Windows: 32 bit)
unsigned long unsigned long int	unsigned long int			
long long, long long int signed long long, signed long long int	long long int	>= 64 bit	64 bit	64 bit
unsigned long long unsigned long long int	unsigned long long int			

Fixed-width integer types

If you need guaranteed sizes, use the types defined `<cstdint>`:

- signed integers: `int8_t`, `int16_t`, `int32_t`, `int64_t` with width of 8, 16, 32, or 64 bits
- unsigned integers: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` with width of 8, 16, 32, or 64 bits
- note: these types are all defined in namespace std.

Caveat: these types are only defined if the C++ compiler directly supports the type!

```cpp
#include <cstdint>

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

Integer literals

Integer literals represent constant values

- decimal: `42`
- octal (base 8): `052` (with `0` prefix)
- hexadecimal (base 16): `0x2a` or `0X42` (with `0x` or `0X` prefix)

Careful about sizes: `0xffff` might be -1 or 65535, depending on the type of integer

- suffixes allow to specify the type
  - `unsigned` suffix: `42u` or `42U`
  - `long` suffix: `42l` or `42L`
  - `long long` suffix: `42ll` or `42LL`
  - combinations are possible, e.g. `42ul`, `42ull`

Single quotes can be inserted between digits as separator

- e.g. `1'000'000'000'000ull`

Logical values

Logical values are of type `bool`

- possible values are `true` and `false`
- size of `bool` is implementation defined
- often obtained from implicit automatic type conversion (see later)

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

Floating point numbers

Floating point numbers (usually) in IEEE-754 format

- `float`: 32 bit floating point (approx. 7 decimal digits)
- `double`: 64 bit floating point (approx. 15 decimal digits)
- `long double`: extended precision floating point, usually between 64 and 128 bit
    - 80 bit on x86/x64 architecture (approx. 19 decimal digits), not necessarily IEEE-754 compliant

Floating point types may take special values

- infinity or -infinity (inf or -inf)
- not-a-number (nan)
- negative zero (-0.0)

Floating point literals

Floating point literals represent constant values

- without exponent: `3.141592`, `.5`
- with exponent: `1e9`, `6.26e-34`

By default, a floating point literal is of type `double`

- suffixes allow to specify the type
  - `float` suffix: `1.0f` or `1.0F`
  - `long double` suffix: `1.0l` or `1.0L`

Single quotes can be inserted between digits as separator

- e.g. `6.626'070e-34`

Characters and strings

Characters and strings are messy in C++, unfortunately.

- character data types:
  - `char`: guaranteed minimum width 8 bit
    - depending on implementation, might be `signed char` or `unsigned char`
    - best to rely only on English alphabetic characters, digits, and basic punctuation characters
  - `char16_t` and `char32_t` to represent UTF-16 and UTF-32 characters
  - `char8_t` to represent UTF-8 characters (since C++20)
  - `wchar_t` is implementation defined...
- string data types: usually represented using the class `std::string`, see later

Character literals

Character literals represent constant values

- any regular character, e.g. `'a'`, `'0'`
- escape sequences, e.g. `'\''`, `'\\'`, `'\n'`, `'\u1234'`

By default, character literals are of type `char`

- prefixes allow to specify the type
  - UTF-8 literal: `u8'a'` for type `char8_t` (since C++20)
  - UTF-16 literal: `u'a'` for type `char16_t`
  - UTF-32 literal: `U'a'` for type `char32_t`

Void type

The type void has no values

- no objects of type void are allowed
- mainly used to indicate return type for functions without a return value

```
void object;            // ERROR: object of type void
void someFunction() {  // ok: void return type
    // ...
}
```

Contents

Variables

Variables always have to be defined before use.

- declaration (and definition): type specifier followed by comma-separated list of declarators (variable names)

```
int a, b;
float bigNumber;
```

- optional: initialization in declaration

```
int a{42};
```

Initialization

There are two kinds of initialization:

- safe: `variableName{<expression>}`
- unsafe: `variableName = <expression>` or `variableName(<expression>)`

Why is one safe and the other unsafe?

- the unsafe version may do (silent) implicit conversions
- the safe version will yield a compiler error (or warning in gcc) if an implicit conversion potentially results in loss of information

Initialization is (unfortunately) optional

- non-local variables are default-initialized (usually zero for built-in types)
- local variables are usually not default-initialized
  - this can lead to undefined behavior when accessing an uninitialized variable

Initialization (cont.)

Examples:

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

Initializers may be arbitrarily complex:

```
double pi{3.1415926}, z{0.30}, a{0.5};
double volume{pi * z * z * a};
```

Contents

Declarations

Any name / identifier has to be declared before use, so the compiler knows what entity this name refers to, for example:

- objects, such as local or global variables
- references or pointers to objects
- functions, templates, types, aliases

A declaration takes the following (simplified) form:

- optional prefix (e.g. `static`, `virtual`)
- base type (e.g. `std::vector<double>`, `const int`)
- declarator with optional name (e.g. `n`, `*(*)[]`)
- optional suffix (e.g. `const`, `noexcept`)
- optional initializer / function body (e.g. `{3}`, `{ return x; }`)

Const qualifier

Any type `T` in C++ can be const-qualified:

- either before type: `const T`
- or after type: `T const` (less popular)
- a `const` object is considered immutable and cannot be modified

Example:

```cpp
const int i{1};   // declare and define i, initialized to 1
i = 2;            // ERROR: i is constant
```

Declarations: examples

```cpp
int i;
```

- declares and defines `i` as type `int`, no initialization

```cpp
auto count = 1;
```

- declares and defines `count`, type is automatically deduced as `int`, initialized to 1

```cpp
const float pi{3.1415926};
```

- declares and defines `pi` as type `const float`, initialized to 3.1415926, cannot be modified

```cpp
std::vector<std::string> people{"Martin", "Markus"};
```

- declares and defines `people` as type `std::vector<std::string>`, initialized to contain the strings `"Martin"` and `"Markus"`

```cpp
extern int errorNumber;
```

- declares `errorNumber` as type `int`, no definition!

Declarations: examples (cont.)

```
struct Date { int d, m, y; };
```

- declares `Date` as a structure containing three members

```
int day(Date p) { return p.d; }
```

- declares and defines the function `day`, taking a `Date` as argument, returning an `int`

```
float sqrt(float);
```

- declares the function `sqrt`, taking a `float` as argument, returning a `float`, no definition!

```
struct User;
```

- declares `User` as a structure (forward declaration)

```
using Cmplx = std::complex<double>;
```

- declares and defines an alias `Cmplx` for type `std::complex<double>`

Names

- names are a sequence of letters and digits
- the first character must be a letter
  - the underscore _ is considered a letter
  - non-local names starting with underscore are reserved
- some names (keywords) are reserved
  - for example the built-in types (`int`, `float`, etc.)
  - other examples are `if`, `else`, `switch`, `auto`, `true`, etc.
- names are case sensitive!
  - for example, `Count` and `count` are different names!
- good names are important (and hard to choose)
  - do not encode the type in the name
  - maintain a consistent naming style (e.g. camelCase)
    - use tools to enforce it (e.g. clang-format)

Type aliases

- you can assign new names (aliases) to existing types with the `using` statement:

```cpp
using myInt = long;
using myVector = std::vector<int>;
```

- this is mostly useful for longer types (e.g. templates, see later)
- an older version of this exists too:

```cpp
typedef int myNewInt; // equivalent to using myNewInt = int;
```

Contents

Scopes

A scope (or block) is marked by curly brackets: `{}`

- a declaration introduces a name into a scope
- the name can only be used within that scope

There are several types of scopes:

- local scope: corresponding to blocks marked by `{}`
- class scope: the scope inside a class declaration, names are called *member names* or *class member names* (more later)
- namespace scope: the scope inside a namespace declaration, names are called *namespace member names* (more later)
- statement scope: anything declared in the `()` part of a `for`, `while`, `if`, `switch` statement; extends until end of statement
- global scope: basically anything outside the other scopes

Scopes: examples

```
int x{0};          // global x

void f1(int y) { // y local to function f1
    int x;         // local x that hides global x
    x = 1;         // assign to local x
    {
        int x;     // hides the first local x
        x = 2;     // assign to second local x
    }
    x = 3;         // assign to first local x
    y = 2;         // assign to local y
    ::x = 4;       // assign to global x
}

int y = x;         // assigns global x to y (also global)
```

- `::` is the scope resolution operator, so you can refer to hidden global names
- local hidden names cannot be referred to

Scopes: nasty examples

```
int x{42};          // global x

void f2() {
    int x = x;      // nasty! initialize local x with its own value
                    // which is uninitialized...
}

void f3() {         // nasty function!
    int y = x;      // local y initialized with global x == 42
    int x = 21;     // local x hiding global x
    y = x;          // local y assigned local x == 21
}

void f4(int x) {    // x local to function f4, hiding global x
    int x;          // ERROR: x already defined!
}

void f5() {         // no name clash due to statement scope!
    for (int i = 0; i < 5; ++i) std::cout << i << "\n";
    for (auto i : {0, 1, 2, 3, 4}) std::cout << i << "\n";
}
```

Lifetimes of objects

The lifetime of an object

- starts when its constructor completes
- ends when its destructor starts executing

- objects of types without a declared constructor (e.g. `int`) can be considered to have default constructors and destructors that do nothing
- using an object outside its lifetime leads to undefined behavior!

Each object also has a storage duration

- which begins when its memory is allocated and ends when its memory is deallocated
- the lifetime of an object never exceeds its storage duration

Storage durations

Each object has one of the following storage durations:

- automatic: allocated at beginning of scope, deallocated when it goes out of scope
    - e.g. local variables, typically allocated on stack
- static: allocated when program starts, lives until end of program
    - global variables, or variables marked with `static`
- dynamic: allocation and deallocation is handled manually (see later)
    - using `new` and `delete`
    - forgetting deallocation leads to memory leaks! (see later)
    - using an object after deallocation is undefined behavior! (see later)
- thread-local: allocated when thread starts, deallocated automatically when thread ends; each thread gets its own copy
    - declared using `thread_local` keyword

Storage durations: examples

```cpp
int foo{1};                 // static storage duration
static int bar{2};          // static storage duration
thread_local int duh{3}; // thread-local storage duration

void f() {
    int x{4};               // automatic storage duration
    static int y{5};        // static storage duration
}
```

Contents

Namespaces

Large projects contain many names (variables, functions, classes)

- namespaces allow grouping into logical units
- helps to avoid name clashes

Example:

```
namespace myName {
    int a;
}
```

- a namespace also provides a named scope
- members can be referred to using `myName::`, i.e. a full qualifier

Namespaces: example

```cpp
namespace A {
    void foo() { /* ... */ }
    void bar() {
        foo();    // refers to A::foo
    }
}

namespace B {
    void foo() { /* ... */ }
}

int main() {
    A::foo();    // calls foo() from namespace A
    B::foo();    // calls foo() from namespace B

    foo();       // ERROR: no foo() declared in this scope
}
```

Nested namespaces

Namespaces can be nested:

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

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

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

Please note: namespaces are open, i.e. you can add names from several namespace declarations (as done above)

Namespaces: practical note

Code is complex and can contain many braces

- ensure readability using comments

```cpp
// ----------------------------
namespace A {
    void foo() {
        // something
    }

    void bar() {
        // something else
    }

} // end namespace A

// ----------------------------
namespace B {
    // more things

} // end namespace B
```

Using namespaces

- fully qualified names are good for readability
- but sometimes it can be tedious or undesired
- the statement `using X::a;` imports the symbol a into the current scope
- the statement `using namespace X` imports all symbols from namespace X into the current scope
    - be careful with this and use sparingly!

```cpp
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 undefined
    B::z = 3; // OK
}
```

Contents

Initialization revisited

There are two kinds of initialization:

- safe:

```
          int a{42};     // OK, a == 42
          int b{7.5};    // ERROR: no narrowing allowed
```

- unsafe:

```
          int a = 42;    // OK, a == 42
          int b = 7.5;   // OK, but b == 7
          int c(42);     // OK, c == 42
          int d(7.5);    // OK, but d == 7
```

Prefer initialization using `{}`!

Initialization subtleties

Initialization using `{}` is of type `std::initializer_list<T>`, where T is the type of the value inside `{}`

- this can lead to surprises:

```
              std::vector<int> v1{99}; // v1 is a vector of 1 element
                                       // with value 99
              std::vector<int> v2(99); // v2 is a vector of 99 elements,
                                       // each with default value 0
```

- or errors:

```
              std::vector<std::string> v1{"hello"};
                 // v1 is a vector with 1 element with value "hello"
              std::vector<std::string> v2("hello");
                 // ERROR: no constructor of vector accepts a string
```

This does not happen often, but it pays off to be aware.

Initializer lists

- As the name implies, `std::initializer_list<T>` can accept more than one value:

```
struct S { int x; std::string s; };
S s{1, "hello"};                        // struct initializer

std::complex<double> z{0, 1};           // uses constructor
std::vector<double> v{0.0, 1.1, 2.2}; // uses list constructor
```

- Again, subtleties apply:

```
std::complex<double> z1(0, 1); // uses constructor
std::complex<double> z2{};     // uses constructor,
                               // default value {0,0}
std::complex<double> z3();     // function declaration!

std::vector<double> v1{10, 3.3}; // list constructor
std::vector<double> v2(10, 3.3); // constructor, v has 10
                                 // elements set to 3.3
```

Default values

Initializing with empty `{}` yields the default value

- for integral types, the default is a suitable representation of 0
- for user-defined types, the default value (if there is any) is determined by the constructor (see later)

```
int x{};      // x == 0
double d{};   // d == 0.0

std::vector<int> v{}; // v is the empty vector
std::string s{};      // s == ""
```

Missing initializers

Unfortunately the initializer is optional, leading to potential undefined behavior

- this is due to performance considerations
- objects with static storage duration are default initialized
- all other objects are not default initialized

```
int a;      // global, i.e. equivalent to int a{}; so a == 0

void f() {
    int x;     // local, not initialized!
    int y = x; // value of y is undefined!
}
```

- This can end up in very hard to find bugs!
  - use compiler warnings (-Wall) and heed them

Auto

- since the type of the initializer is known to the compiler, it can save you some work
- you can replace the type in a declaration by the keyword auto

```cpp
auto a{123};   // a is of type int
auto b{'c'};   // b is of type char
```

- this is most useful for complicated types, such as

```cpp
std::unique_ptr<int> u1{std::make_unique<int>()};
auto u2{std::make_unique<int>()}; // same as u1

std::vector<double> v{};
std::vector<double>::iterator i1 = v.begin();
auto i2 = v.begin(); // same as i1
```

Auto: how to know what happens

- use `type_index` from library boost:

```cpp
#include <iostream>
#include <memory>
#include <boost/type_index.hpp>
using namespace boost::typeindex;

int main() {
    auto a = {13, 14};
    auto u{std::make_unique<int>()};

    std::cout << "Type a: " <<
      type_id_with_cvr<decltype(a)>().pretty_name() << "\n";
    std::cout << "Type u: " <<
      type_id_with_cvr<decltype(u)>().pretty_name() << "\n";
}
```

- easiest to do in online tools such as `https://wandbox.org`:

```
Type a: std::initializer_list<int>
Type u: std::unique_ptr<int, std::default_delete<int> >
```

Contents

Arrays

For arrays with a fixed number of elements, use `std::array<T, num>`

- `num` elements of type `T` that lie contiguously in memory
- can be default initialized using `{}`, or list initialized
- size is accessible via member function `size()`
- access to elements using
  - `operator[]()` (not bounds-checked)
  - `at()` (bounds-checked)
- has iterators for iterating over all elements

Array: example

```cpp
#include <array>
#include <iostream>

int main() {
    std::array<int, 10> a{}; // array of 10 int, default initialized
    std::array<float, 3> b{0.0, 1.1, 2.2}; // array of 3 float

    for (unsigned i = 0; i < a.size(); ++i)
        a[i] = i + 1; // no bounds checking

    for (auto i : b) // loop over all elements of b
        std::cout << i << "\n";

    a.at(11) = 5; // run-time error: out_of_range exception
}
```

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 11) >= _Nm (which is 10)
0
1.1
2.2
```

C-style arrays: DO NOT USE

There are old C-style arrays, for example

```
          int a[10] = {};
          float b[3] = {0.0, 1.1, 2.2};
```

DO NOT USE THEM! THEY ARE EVIL!

- they only exist for C-style compatibility
- they don't know their own size, they have no bounds checking
- they are the source for many crashes, buffer overflows etc.

`std::array` can do everything C-style arrays can

- in a safe way with no overhead
- while providing additional functionality (such as bounds checking, iterators)

Non-fixed size arrays: vectors

If you need arrays that are not fixed size (like `std::array`), use `std::vector<T>`

- dynamically sized array, storage is automatically expanded and contracted as needed
- still guaranteed to be contiguously in memory
- same interface as `std::array`
- and additional functions, such as
    - `push_back` to insert elements at end
    - `clear` to clear the contents
    - `resize` to resize the vector

Vector: example

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

int main() {
    std::vector<int> a;  // default initialized to be empty
    for (unsigned i = 0; i < 10; ++i)
        a.push_back(i + 1);

    std::cout << "Size: " << a.size() << "\n";
    a.clear();
    std::cout << "Size: " << a.size() << "\n";

    a.resize(10); // a now contains 10 zeros
    std::cout << "Size: " << a.size() << "\n";

    for (unsigned i = 0; i < a.size(); ++i)
        a[i] = i + 1; // no bounds checking
}
```

Output:

```
Size: 10
Size: 0
Size: 10
```

Range for loops

Ranges (e.g. containers that support iterators, like `std::array` or `std::vector`) support special range for loops:

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

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

Range for loop: example

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

int main() {
    std::vector<int> a{1, 2, 3, 4, 5}; // initializer list

    for (unsigned i = 0; i < 5; ++i)   // regular for loop
        a.push_back(i + 10);

    for (const auto& e : a)     // range for loop
        std::cout << e << ", ";
    std::cout << "\n";

    for (auto i : {47, 11, 3}) // range for loop
        std::cout << i << ", ";
}
```

Output:

```
1, 2, 3, 4, 5, 10, 11, 12, 13, 14,
47, 11, 3,
```

Contents

Expressions

An expression is a sequence of operators and operands

- evaluation of the expression can produce a result
  - for example, evaluation of `2+3` produces the result `5`
- evaluation of the expression may produce side effects
  - for example, evaluation of `std::cout << "4"` prints `4` on the standard output

lvalues and rvalues

Each expression is characterized by two independent properties

- its type (e.g. `int`, `float`)
- its value category

There are several value categories, extremely simplified there is:

- lvalues that refer to the identity of an object
  - modifiable lvalues can be used on left-hand side of assignment
- rvalues that refer to the value of an object
  - lvalues and rvalues can be used on right-hand side of assignment

More on this later.

Operators

Operators act on a number of operands

- unary operators
  - such as: negation (`-`), address-of (`&`), dereference (`*`)
- binary operators
  - such as: equality (`==`), multiplication (`*`)
- ternary operator: `a ? b : c`

Most operators can be overloaded for user-defined types (see later).

Operands

The operands of any operator can be

- other expressions
- or primary expressions

Primary expressions are

- literals
- variable names
- and others (lambdas, fold expressions, see later)

Arithmetic operators

Operator	Explanation
`+a`	unary plus
`-a`	unary minus
`a + b`	addition
`a - b`	subtraction
`a * b`	multiplication
`a / b`	division
`a % b`	modulo
`~a`	bit-wise NOT
`a & b`	bit-wise AND
`a | b`	bit-wise OR
`a ^ b`	bit-wise XOR
`a << b`	bit-wise left shift
`a >> b`	bit-wise right shift

Undefined behavior can occur, e.g. on

- signed overflow
- division by zero
- shift by negative offset
- shift by offset larger than the width of the type

Logical and relational operators

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

Assignment operators

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`	bit-wise AND assignment
`a |= b`	bit-wise OR assignment
`a ^= b`	bit-wise XOR assignment
`a <<= b`	bit-wise left shift assignment
`a >>= b`	bit-wise right shift assignment

- left-hand side of assignment op. must be 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 (cont.)

Assignment operators return a reference to the left-hand side:

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

Very rarely used, except in these situations:

```cpp
if (int d = computeValue()) {
    // executed if d is not zero
}
else {
    // executed if d is zero
}
```

Increment and decrement operators

Operator	Explanation
`++a`	prefix increment
`--a`	prefix decrement
`a++`	postfix increment
`a--`	postfix decrement

Logic differs between prefix and postfix variants:

- prefix variants increment/decrement the value of an object and return a reference to the result
- postfix variants create a copy of an object, increment/decrement the value of the original object, and return the unchanged copy

Ternary conditional operator

Operator	Explanation
`a ? b : c`	conditional operator

Semantics:

- a is evaluated and converted to `bool`
- if result was `true`, b is evaluated
- if result was `false`, c is evaluated

```
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

- 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
- grouping is often not immediately obvious
  - use parentheses judiciously!

Precedence and associativity do not specify evaluation order

- evaluation order is mostly unspecified
- in general, it is undefined behavior to refer to and change the same object within one expression

Precedence and associativity (cont.)

Precedence can be obvious:

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

But it can get confusing very fast:

```cpp
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
```

Bugs like to hide in expressions without parentheses:

```cpp
if (0 <= x <= 99) // not what you might expect!
    std::cout << "I am always true!";

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

Operator precedence table

Prec.	Operator	Description	Associativity
1	`::`	scope resolution	left-to-right
	`a++` `a--`	postfix increment/decrement	
	`<type>() <type>{}`	functional cast	
2	`a()`	function call	left-to-right
	`a[]`	subscript	
	`.` `->`	member access	
	`++a` `--a`	prefix increment/decrement	
	`+a` `-a`	unary plus/minus	
	`!` `~`	logical/bit-wise 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 (cont.)

Prec.	Operator	Description	Associativity
4	`.*` `->*`	pointer to member	left-to-right
5	`a*b` `a/b` `a % b`	multiplication / division / modulo	left-to-right
6	`a+b` `a-b`	addition / subtraction	left-to-right
7	`<<` `>>`	bit-wise shift	left-to-right
8	`<=>`	three-way comparison (C++20)	left-to-right
9	`<` `<=`	relational $<$ and $\leq$	left-to-right
	`>` `>=`	relational $>$ and $\geq$	
10	`==` `!=`	relational $=$ and $\neq$	left-to-right

Operator precedence table (cont.)

Prec.	Operator	Description	Associativity
11	`&`	bit-wise AND	left-to-right
12	`^`	bit-wise XOR	left-to-right
13	`|`	bit-wise OR	left-to-right
14	`&&`	logical AND	left-to-right
15	`||`	logical OR	left-to-right
	`a ? b : c`	ternary conditional	
	`throw`	throw operator	
	`=`	direct assignment	
16	`+=` `-=`	compound assignment	right-to-left
	`*=` `/=` `%=`	compound assignment	
	`<<=` `>>=`	compound assignment	
	`&=` `^=` `|=`	compound assignment	
17	`,`	comma	left-to-right

Contents

Simple statements

- declaration statement: declaration followed by semicolon

```
int i = 0;
```

- expression statement: any expression followed by a semicolon

```
i + 3;    // valid, but rather useless expression statement
foo();    // valid and possibly useful expression statement
```

- compound statement: brace-enclosed sequence of statements

```
{                 // start of block and scope
    int j = 1;    // declaration statement
    int k = 2;    // declaration statement
}                 // end of block and scope
```

If statement

Conditionally execute another statement:

```
        if (init_statement; condition)
            then_statement
        else
            else_statement
```

- if condition evaluates to true after conversion to bool, then_statement is executed, otherwise else_statement is executed
- both init_statement and 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 (cont.)

`init_statement` is useful for local variables used only inside `if` :

```cpp
if (auto value{computeValue()}; value < 42) {
    // do something
}
else {
    // do something else
}
```

This is equivalent to:

```cpp
{
    auto value{computeValue()};
    if (value < 42) {
        // do something
    }
    else {
        // do something else
    }
}
```

If statements (cont.)

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

```
// INTENTIONALLY MISLEADING!
if (condition0)
    if (condition1)
        // do something if (condition0 && condition1) == true
else
    // do something if condition0 == false (...not!)
```

If 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

Conditionally transfer control to one of several statements:

```
switch (init_statement; condition)
    statement
```

- condition is an expression or single declaration that is convertible to an enumeration or integral type
- body of switch statement may contain 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 (cont.)

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 (cont.)

Less regular example:

```
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;
}
```

- the body is executes sequentially until a `break;` statement is encountered, i.e. it can "fall through"
- compilers may generate warnings when encountering fall-through behavior
  - use special `[[fallthrough]];` statement to mark intentional fall through

While loop

Repeatedly execute a statement:

```
while (condition)
    statement
```

- executes `statement` repeatedly until the value of `condition` becomes `false`
  - `condition` is evaluated before each iteration
- `condition` is an expression that can be converted to `bool` or a single declaration
- `statement` is an arbitrary statement
- the `break;` statement can be used to exit the loop
- the `continue;` statement can be used to skip the remainder of the body

Do-while loop

Repeatedly execute a statement:

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

- executes `statement` repeatedly until the value of `condition` becomes `false`
  - `condition` is evaluated after each iteration
- `condition` is an expression that can be converted to `bool` or a single declaration
- `statement` is an arbitrary statement
- the `break;` statement can be used to exit the loop
- the `continue;` statement can 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:

```cpp
int i{42};

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

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

For loop

Repeatedly executes a statement:

```
        for (init_statement; condition; iteration_expression)
            statement
```

- executes init_statement once, then, if condition is true, executes statement and iteration_expression until condition becomes false
- init_statement is an expression or declaration
- condition is an expression that can be converted to bool or a single declaration
- iteration_expression is an arbitrary expression
- all three statements in the parentheses can be omitted
- the break; statement can be used to exit the loop
- the continue; statement can be used to skip the remainder of the body

For loop (cont.)

Example:

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

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

Careful 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
}
```

Contents

Functions in C++

- functions associate a sequence of statements (function body) with a name
- functions can have zero or more function parameters

```
          return_type name ( parameter_list ) {
              statements (body)
          }
```

- functions are invoked via the function-call expression
- parameters are initialized from the provided arguments

```
              name ( argument_list );
```

Function return types

- no return type is marked using `void`:

```cpp
void foo(int parameter0, float parameter1) {
    // do something with parameter0 and parameter1
}
```

- functions with a return type (non-void) must contain a `return` statement:

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

- the main-function of a program returns an `int`, as an exception the `return` statement may be omitted (resulting in an implicit `return 0;`)

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

Argument passing

- arguments are passed by value:

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

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

- arguments can also explicitly be passed by reference, see next section
- this distinction is essential for user-defined types! (see later)

Unnamed arguments

- function parameters can be unnamed, which means they cannot be used:

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

- the argument still has to be supplied on function invocation:

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

Default arguments

- the last parameters of a function can have default values
    - after a parameter with a default value, all following 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
}
```

Contents

Reference declaration

a reference declaration declares an alias to an existing object

- Lvalue reference: `&declarator`
- Rvalue reference: `&&declarator`
- `declarator` can be any other declarator, except another reference declarator
    - most of the time, `declarator` is just a name

References are special:

- there are no references to `void`
- references are immutable (although the referenced object can be mutable)
- references are not objects, i.e. they do not necessarily occupy storage
    - hence there are no references (or pointers) to references
    - and there are no arrays of references

Reference declaration (cont.)

- the `&` or `&&` qualifiers are part of the declarator, not the type:

```cpp
int i{10};
int &j{i}, k{i}; // j is reference to int, k is int
```

- we can insert or omit whitespaces before and after the `&` or `&&` qualifiers

```cpp
int &m{i}; // valid
int& n{i}; // also valid
```

- by convention we use `int& n{i};`
    - to avoid confusion, statements should only declare one identifier at a time
    - very rarely, exceptions to this rule are necessary (e.g. in the init statements of `if`, `switch`, `for`)

Reference initialization

- a reference to type `T` must be initialized to refer to a valid object
    - an object of type `T`
    - a function of type `T`
    - an object implicitly convertible to `T`
- there are exceptions: (as always...)
    - function parameter declarations
    - function return type declarations
    - class member declarations (see later)
    - when using `extern` modifier

Lvalue references: examples

Alias for existing objects:

```cpp
int i{10};
int j{42};
int& 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: examples (cont.)

Pass by reference for function calls:

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

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

Lvalue references: examples (cont.)

Turning a function call into an lvalue expression:

```cpp
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: examples

Rvalue references cannot (directly) bind to lvalues:

```
int i{10};
int&& j{i};  // ERROR: cannot bind rvalue ref to lvalue
int&& k{42}; // OK
```

Rvalue references can extend the lifetime of temporary objects:

```
int i{10};
int j{32};

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

Rvalue references: examples (cont.)

Overload resolution (see later) allows 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&&)
}
```

Const references

- references themselves cannot be `const`
- however, the reference type can be `const`
- a reference to `T` can be initialized from a type that is not `const`
    - 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 qualifiers
j = 42;     // ERROR: assignment of read-only reference
```

- lvalue references to `const` also extend lifetime of temporaries:

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

Dangling references

- Caution: you can write programs where the lifetime of the referenced object ends while references to it still exist!
  - happens mostly when referencing objects with automatic storage duration
  - results in dangling reference and undefined behavior
- example:

```cpp
int& foo() {
    int i{42};
    return i;    // MISTAKE: returns dangling reference!
}
```

- good compilers warn you about it
  - so make sure to use -Wall or /Wall
  - and heed all compiler warnings!

Contents

Converting between types

Sometimes, the automatic implicit conversion is not enough

- explicit type conversion can be forced to cast between related types

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

  - converts the value of `expression` to a value of `new_type`
  - `new_type` must have same `const`-ness as the type of `expression`
  - many use cases (but never use it lightly!)

- several more casting operators exist (`const_cast`, `reinterpret_cast`, C-style cast: `(new_type) expression`)
  - do not use them! they are evil!
- `dynamic_cast` is sometimes used for polymorphic class hierarchies, see later

static_cast example

```cpp
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
}
```

Summary

Contents

Return types

- we already know how to specify a return type:

```
int foo(); // foo returns an int
```

- C++ also allows a trailing return type:

```
auto foo() -> int; // foo returns an int
```

  - the keyword `auto` is fixed, the actual return type follows after the parameter list and the symbols ->

- this really becomes useful with templates (see later), where the return type depends on the arguments:

```
template <typename T, typename U>
auto sum(const T& x, const U& y) -> decltype(x+y);
```

  - but you might also just like the style better

Returning multiple values

- returning more than one value from a function is not supported directly by the syntax
- but we can use structured bindings and `std::pair` or `std::tuple` to do so:

```cpp
std::pair<int, std::string> foo() {
    return std::make_pair(17, "C++");
}

std::tuple<int, int, float> bar() {
    return std::make_tuple(1, 2, 3.0);
}

int main() {
    auto [i, s] = foo(); // i is int with i == 17,
                         // s is std::string with s == "C++"
    auto [a, b, c] = bar(); // a, b are int, c is float,
                            // a == 1, b == 2, c == 3.0
}
```

Structured bindings

- structured bindings allow you to initialize multiple entities by elements / members of an object (such as `std::pair` or `std::tuple`)
- they work nicely with the standard library, for example with associative containers like `std::map`:

```cpp
std::map<std::string, int> myMap; // map with strings as keys
// ... fill the map ...

// iterate over the container using range-for loop
for (const auto& [key, value] : myMap)
    std::cout << key << ": " << value << "\n";
```

Structured bindings (cont.)

- you can also bind `struct` members or `std::array` entries:

```cpp
struct myStruct { int a{1}; int b{2}; };
auto [x, y] = myStruct{}; // x, y are int, x == 1, y == 2

std::array<int, 3> myArray{47, 11, 9};
auto [i, j, k] = myArray; // i == 47, j == 11, k == 9
```

- structured bindings can have qualifiers (references, `const`):

```cpp
myStruct ms;
auto& [u, v] = ms;  // u, v now reference ms.a, ms.b
ms.a = 11;
std::cout << u;     // prints 11
u = 22;
std::cout << ms.a;  // prints 22
```

- you can even provide a `std::tuple`-like API for your own data types to enable structured bindings (see reference)

Returning multiple values revisited

- specifying the return type for multiple values can be annoying:

```cpp
std::tuple<int, int, float> bar() {
    return std::make_tuple(1, 2, 3.0);
}
```

- with auto we can let the compiler deduce the return type automatically (even without trailing return type):

```cpp
auto bar() {
    return std::make_tuple(1, 2, 3.0);
}
```

Contents

Parameter passing revisited

We can pass parameters to functions:

- by value:

```
void foo(int value);
```

- by reference:

```
void foo(int& value);
```

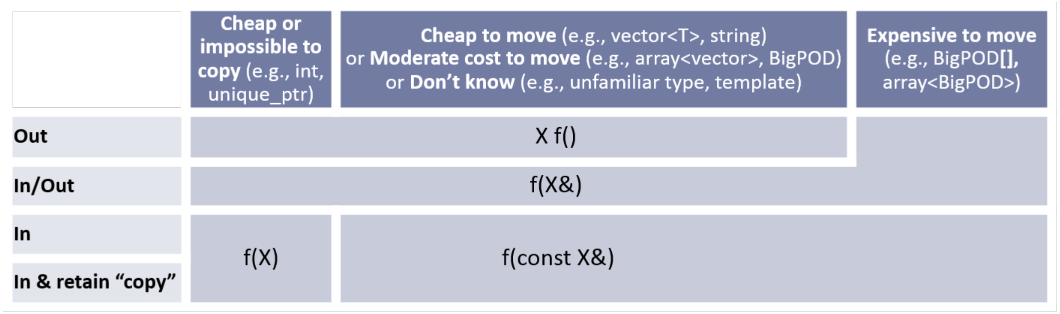
- by const reference:

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

How to choose?

Parameter passing

Refer to the C++ Core Guidelines:
`https://isocpp.github.io/CppCoreGuidelines/`



	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()	X f()	f(X&)
In/Out	f(X&)	f(X&)	f(X&)
In	f(X)	f(const X&)	f(const X&)
In & retain "copy"	f(X)	f(const X&)	f(const X&)

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

* *or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Parameter passing (cont.)

Summarized guidelines:

- “in” parameters: pass by value (for cheaply-copied types) or pass by const reference

```
void f1(const std::string& s); // OK, pass by const reference
void f2(std::string s);        // potentially expensive
void f3(int x);                // OK, cheap
void f4(const int& x);         // bad, unnecessary overhead
```

- “in-out” parameters: pass by reference (if you cannot avoid it)

```
void update(Record& r);  // assume that update writes to r
```

- “out” parameters: return them, either as single return type or as `std::pair`, `std::tuple`

Contents

Overloading

Overloaded functions:

- we can declare different functions having the same name but different argument types:

```
void f(int);    // a function called f, taking an int
void f(double); // another function f, taking a double
```

- on a function call, the compiler automatically resolves the overloads in the current scope and calls the best match
  - if there is no best match, it's a compile error

Overload resolution criteria

The following criteria are tried in order:

1. exact match (no or only trivial conversions, e.g. `T` to `const T`)
2. match using promotions (e.g. `bool` to `int`, `char` to `int`, or `float` to `double`)
3. match using standard conversions (e.g. `int` to `double`, `double` to `int`, or `int` to `unsigned int`)
4. match using user-defined conversions (see later)
5. match using ellipsis `...` (see later)

Overloading: examples

```cpp
void print(int);
void print(double);
void print(long);
void print(char);

void f(char c, int i, short s, float f) {
    print(c);   // exact match: print(char)
    print(i);   // exact match: print(int)
    print(s);   // integral promotion: print(int)
    print(f);   // float to double promotion: print(double)

    print('a'); // exact match: print(char)
    print(49);  // exact match: print(int)
    print(0);   // exact match: print(int)
    print(0L);  // exact match: print(long)
}
```

Overloading: examples (cont.)

```cpp
using complex = std::complex<double>;

int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, complex);

void h(complex z) {
    auto i  = pow(2, 2);     // invokes pow(int, int)
    auto d  = pow(2.0, 2.0); // invokes pow(double, double)
    auto z2 = pow(2, z);     // invokes pow(double, complex)
    auto z3 = pow(z, 2);     // invokes pow(complex, int)
    auto z4 = pow(z, z);     // invokes pow(complex, complex)
    auto e  = pow(2.0, 2);   // ERROR: ambiguous
}
```

Overloading and the return type

Caution: return types are **not** considered in overload resolution!

- this is good, so function calls are context-independent:

```
float sqrt(float);
double sqrt(double);

void f(float fla, double da) {
    float fl{sqrt(da)};  // invokes sqrt(double)
    auto d{sqrt(da)};    // invokes sqrt(double)
    fl = sqrt(fla);      // invokes sqrt(float)
    d = sqrt(fla);       // invokes sqrt(float)
}
```

Contents

Functors

Functions are not objects in C++

- they cannot be passed as parameters
- they cannot have state

However, a type `T` can be a function object (or functor), if:

- `T` is an object
- `T` defines `operator()`

Function objects can be used like functions.

Functor example

Functor storing a state:

```cpp
struct Adder {
    int value{1};

    int operator() (int param) {
        return param + value;
    }
};

int main() {
    Adder myAdder;
    myAdder.value = 5;
    myAdder(1);          // returns 6
    myAdder(4);          // returns 9
    myAdder.value = 7;
    myAdder(1);          // returns 8
}
```

std::function

std::function is a wrapper for all callable targets

- defined in <functional> header
- stores, copies, and invokes the wrapped target
- caution: can incur a slight overhead in both performance and memory

```cpp
#include <functional>
int addFunc(int a) { return a + 3; }

int main() {
    std::function adder{addFunc};
    int a{adder(5)};   // a == 8

    // alternatively specifying the function type:
    std::function<int(int)> adder2{addFunc};
}
```

- function type is declared as `return_type(argument_list)`
- deduction guides usually makes this unnecessary

std::function example

```cpp
#include <functional>
#include <iostream>

void printNum(int i) { std::cout << i << "\n"; }

struct PrintNum {
    void operator() (int i) { std::cout << i << "\n"; }
};

int main() {
    // store a function
    std::function f_printNum{printNum};
    f_printNum(-47);

    // store the functor
    std::function f_PrintNum{PrintNum{}};
    f_PrintNum(11);

    // fix the function parameter using std::bind
    std::function<void()> f_leet{std::bind(printNum, 31337)};
    f_leet();
}
```

Contents

Lambda expressions

Lambda expressions are a simplified notation for anonymous function objects

- they are an expression and can be used anywhere expressions can be used:

```
std::find_if(container.begin(), container.end(),
    [](int val) { return 1 < val && val < 10; }
);
```

- the function object created by the lambda expression is called closure
- the closure can hold copies or references of captured variables

Lambda expressions: the syntax

```
         [ capture_list ] ( param_list ) -> return_type { body }
```

- `capture_list` specifies the variables of the environment to be captured in the closure
- `param_list` are the function parameters
- `return_type` specifies the return type; it is optional! If not specified, the return type is deduced from the return statements in the body

- the `capture_list` can be empty: `[]() {}`
- if the `param_list` is empty, it can be omitted
  - the shortest lambda expression is thus: `[]{}`

Lambda expressions: examples

```
bool isMultipleOf3(int v) { return v % 3 == 0; }

void f() {
    std::vector<int> numbers{ /* ... */ };

    // version with functions:
    std::remove_if(numbers.begin(), numbers.end(), isMultipleOf3);

    // version with lambdas:
    std::remove_if(numbers.begin(), numbers.end(),
        [](int v) { return v % 3 == 0; }
    );
}
```

Storing lambda expressions

Lambda expressions can be stored in variables:

- using auto:

```cpp
auto lambda = [](int a, int b) { return a + b; };

std::cout << lambda(2, 3) << "\n"; // outputs 5
```

- using std::function:

```cpp
std::function func = [](int a, int b) { return a + b; };

std::cout << func(3, 4) << "\n"; // outputs 7
```

- the signature of the lambda can be stated explicitly:

```cpp
std::function<int(int, int)> func2 =
    [](int a, int b) { return a + b; };

std::cout << func2(4, 5) << "\n"; // outputs 9
```

The type of a lambda expression

- the type of a lambda expression is not defined
- no two lambdas have the same type (even if they are identical otherwise):

```cpp
auto myFunc(bool first) { // ERROR: ambiguous return type
    if (first)
        return []() { return 42; };
    else
        return []() { return 42; };
}
```

- for each lambda, the compiler generates a unique closure class with a constructor and `operator() const`
  - in fact, we could do this ourselves, it's just more effort
  - nevertheless, lambdas turned out to be a game changer

Lambda expression return types

- just like for functions, the return type can be deduced from the return statement:

```
void f() {
    auto x = [] { std::cout << "Hello\n"; }; // void return type

    auto y = [] { return 42; }; // int return type

    auto z = [] { if (true) return 1; else return 2.0; };
                  // ERROR: inconsistent types

    auto z2 = [] -> int { if (true) return 1; else return 2.0; };
                  // OK: explicit return type
}
```

Lambda captures

- lambda captures are part of the state of a closure
    - can refer to automatic variables in the surrounding scopes
    - can refer to the `this` pointer in the surrounding scope (see later)

- capture can be done by copy: creates a copy of the captured variable in the closure
- capture can be done by reference: creates a reference to the captured variable in the closure

- captures can be used in the lambda expressions like regular variables or references

Lambda captures: examples

```cpp
void f() {
    int i{0};

    auto lambda1 = [i]() { std::cout << i; }; // i by copy
    auto lambda2 = [i]() { ++i; }; // ERROR: i is read-only!

    auto lambda3 = [&i]() { ++i; }; // OK: i by reference
    lambda3();
    std::cout << i; // outputs 1
}
```

Caution: beware dangling references:

```cpp
auto g() {
    int j{0};
    return [&j]() { ++j; }; // beware: reference to j will dangle!
}
```

Lambda captures: examples (cont.)

Capture by copy vs. by reference:

```
void f() {
    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 default captures

- capture defaults allow you to capture all variables in the surrounding scope
  - by copy: `[=]`
  - by reference: `[&]`

- if defaults are used, only diverging capture types can be specified afterwards:

```cpp
void f() {
    int i{0}, j{42};

    auto lambda1 = [&, i]() {};  // j by reference, i by copy
    auto lambda2 = [=, &i]() {}; // j by copy, i by reference

    auto lambda3 = [&, &i]() {}; // ERROR: non-diverging capture
    auto lambda4 = [=, i]() {};  // ERROR: non-diverging capture
}
```

Lambda default captures (cont.)

- default captures can be dangerous
- in particular default by reference `[&]` may have unwanted side effects

- in general: avoid default captures!

Contents

Handling error conditions

What to do in error conditions?

- terminate the program: `if (somethingWrong) exit(1);`
  - quite drastic, very problematic in libraries
- return an error value and let the caller decide
  - often hard to define an error value, e.g. `int getInt();`
- return a legal value and leave program in error state
  - e.g. the C standard library sets global variable `errno`:

```
double d = sqrt(-1.0);  // value of d is meaningless
```

  - need to test `errno` basically everywhere, also issues with concurrency and global `errno` variable
- call an error-handler: `if (wrong) errorHandler();`
  - but how can the handler handle the problem?

Exceptions

The C++ “solution” is exceptions:

- exceptions transfer control and information up the call stack
- see if the caller(s) can handle the exceptional condition

- exceptions are raised via `throw` expressions
  - `dynamic_cast`, `new` and some standard library functions can also raise exceptions
- exceptions are handled in `try`-`catch` blocks
  - handling them is optional
  - however, if an exception is not caught, the program is terminated
  - errors during handling an exception also lead to program termination

Throwing exceptions

- objects of any type may be thrown as exception objects
- syntax: `throw expression;`
- copy initializes the exception object from `expression` and throws it
- the standard library offers `std::exception` and some derived exception types such as `std::invalid_argument` or `std::out_of_range`
  - `std::exception` contains an explanatory string, it can be queried using `what()`
  - your own exceptions should usually derive from the standard library classes

```cpp
#include <stdexcept>

void foo(int i) {
    if (i == 42)
        throw 42;

    throw std::logic_error("What is the meaning of life?");
}
```

Handling exceptions

- when an exception is thrown, C++ performs stack unwinding
    - ensures proper clean up of objects with automatic storage duration
    - there is some slight run-time overhead, so exceptions are often avoided in real-time applications
- the stack is unwound until a `try`-`catch` block is found
    - exceptions that occur in the `try` block can be handled in the `catch` block
    - the parameter of the `catch` block determines the type of exception that causes the block to be entered

Exception handling example

```cpp
#include <exception>
#include <iostream>

void bar() {
    try {
        foo(42);
    } catch (int i) {
        /* handle the exception somehow */
    } catch (const std::exception& e) {
        std::cerr << e.what() << "\n";
        /* handle the exception somehow */
    }
}
```

Noexcept functions

- some functions do not throw (or should not throw)
- you can mark functions as `noexcept`:

```
int myFunction() noexcept; // may not throw an exception
```

- valuable information for the programmer (no need to handle exceptions) and the compiler (optimizations)
- however, the compiler cannot fully check for the compliance of `noexcept` functions
  - if a `noexcept` function throws anyway, the program is terminated immediately
  - this can happen unexpectedly, e.g. using a `std::vector` and not having enough memory (leading to a `std::bad_alloc` exception)

Exception guidelines

- exceptions should be used rarely
    - main use case: establishing class invariants, for example in RAII (see later)
- 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

Contents

Classes

- a class is a user-defined type used for data abstraction
- class definition:

```
class_keyword name {
    member_specifications
};
```

- `class_keyword` is either `class` or `struct`
- `name` is any valid identifier
- `member_specifications` is a list of declarations, mainly:
    - variables or data members
    - functions or member functions
    - types or nested types
- the trailing semicolon is mandatory!

Data members

- data members are simply variable declarations inside a class
  - the name must differ from the class name
  - they can be initialized, if desired (it's recommended!)
- the declaration must be a complete type (so size of storage is known, see later)
- there are some restrictions on modifiers:
  - `extern` is not allowed
  - `static` members are allowed (see later), they can additionally be `thread_local`

```cpp
struct Foo {
    int i{42};
    std::string s{"Hello"};
    std::vector v; // not initialized

    static int magicValue; // static member variable
};
```

Accessing class members

- class members are accessed via operator `.` for objects:

```cpp
struct Bar {
    int i{42};
    const float f{3.14};
};

Bar b;     // variable (or instance) b of class Bar
b.i += 42; // now b.i == 84
std::cout << b.f << "\n"; // output 3.14
```

- for pointers to an object (see later) we can use operator `->`

```cpp
Bar  c;      // instance c of class Bar
Bar* p = &c; // pointer p pointing to c
// ^ (don't actually do this, it's just for illustration here)
std::cout << (*p).i << p->i << "\n"; // output c.i twice
```

Member functions

- member functions are function declarations inside a class
- modifiers: member functions can be `static` (see later)
- qualifiers: non-static member functions can be qualified
  - with `const` (see later)
  - ref-qualified (see later)
- non-static member functions can be `virtual` (see later)
- there are some special member functions (guess what, there is a theme here: see later!)
  - constructors, destructor
  - overloaded operators

```
struct Foo {
    void foo(); // non-static member function
    void cfoo() const; // const-qualified non-static member fct.
    static void bar(); // static member function
    Foo(); // constructor
    ~Foo(); // destructor
    bool operator==(const Foo& f); // overloaded operator==
};
```

Defining member functions

- member functions can be defined
  - inside the class (inline)
  - outside the class (out-of-line)
- if defined out-of-line, the qualifiers (e.g. `const`) need to match the declaration

```cpp
struct Foo {
    void foo1() { /* ... */ } // inline definition
    void foo2();
    void fooConst() const;
    static void fooStatic();
};
// out-of-line definitions:
void Foo::foo2() { /* ... */ }
void Foo::fooConst() const { /* ... */ }
void Foo::fooStatic() { /* ... */ }
```

- the class declaration (and inline definitions) is usually put in header files (.h / .hpp)
- the out-of-line definitions are usually put in source files (.cpp)

Accessing data members in member functions

- non-static member functions can access non-static members without explicit qualification (preferred!)
- non-static member functions have an implicit parameter `this`
    - in member functions without qualifier or with ref qualifier `this` is of type `C*`
    - in const qualified member functions `this` is of type `const C*`

```cpp
struct C {
    int i;
    int foo() {
        this->i; // explicit member access, 'this' is of type C*
        return i; // implicit member access (preferred)
    }

    int foo() const {
        return this->i; // 'this' is of type const C*
        // preferred would be: return i;
    }
};
```

Nested types

- user-defined types can be nested
  - the surrounding class behaves like a namespace
  - the nested type is accessed via scope resolution operator ::
- the nested type has full access to the parent (friend, see later)

```cpp
struct A {
    struct B {
        int doStuff(const A& a) {
            return a.i; // OK, B is automatically friend of A
        }
    };

private:
    int i;
};

A::B b; // instantiate nested type B of class A
```

Access control

- every member of a class has access control:
  - `public` members can be accessed by everyone
  - `protected` members can be accessed by the class itself and its subclasses
  - `private` members can only be accessed by the class itself
- classes defined with
  - `class` have default access `private`
  - `struct` have default access `public`

```cpp
class Foo {
    int a; // a is private by default
public: // everything after this is public
    int b;
    int getA() const { return a; }
protected: // everything after this is protected
    int c;
public: // everything after this is public
    int ml{42};
};
```

Friends of classes

- you can put friend declarations in the class body, granting the friend access to private and protected members of the class
- friends can be declared as:
  - `friend function_declaration ;` declaring a non-member function as friend of the class (with an implementation elsewhere)
  - `friend function_definition ;` defines a non-member (!) function and declares it as friend of the class
  - `friend class_specifier ;` declares another class as friend of the class

- friendship is non-transitive and cannot be inherited
- it does not matter where friends are declared (i.e. they can appear in `public` or `private` sections)

Friends: example

```
class A {
    int a;
    friend class B; // class B is friend of A
    friend void foo(A&); // non-member function foo is friend of A

    friend void bar(A& a) { a.a = 42; }
    // OK, non-member function bar is friend of A
};

class B {
    friend class C; // class C is friend of B
    void foo(A& a) { a.a = 42; } // OK, B is friend of A
};

class C {
    void foo(A& a) { a.a = 42; } // ERROR, C is not friend of A
};

void foo(A& a) { a.a = 42; } // OK, foo is friend of A
```

Forward declarations and incomplete types

- classes can be forward declared:

```
class_keyword name ;
```

- this declares an incomplete type that will be defined later in the scope
- incomplete types can be used for references / pointers
- this can be useful for:
  - breaking circular dependencies between classes
  - to reduce compilation times by avoiding transitive includes of expensive-to-compile headers (often done in standard library headers)

Forward declaration: example

Header foo.h:

```cpp
// forward declarations
class A;
class ClassFromExpensiveHeader;

class B {
    ClassFromExpensiveHeader& member;

    void foo(A& a);
};

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

Source foo.cpp:

```cpp
#include "ExpensiveHeader.h"

// ... implementation ...
```

Const correctness

- member functions can be qualified as `const` (which changes their type for overload resolution)
- in const functions the `this` pointer is `const`, meaning you cannot change the state of its object

```cpp
class A {
    int i{42};
public:
    int get() const { return i; } // const getter
    void set(int j) { i = j; } // non-const setter
};

void foo(A& nonConstA, const A& constA) {
    int a = nonConstA.get(); // OK
    int b = constA.get();    // OK
    nonConstA.set(11); // OK, non-const
    constA.set(12);    // ERROR: non-const method on const object
}
```

- `const` can be “circumvented” using `mutable` (not recommended, see reference)

Const overloads

- const is used for overload resolution
  - non-const lvalues prefer non-const overloads over const ones
  - const lvalues only use const overloads

```cpp
struct Foo {
    int getA()       { return 1; }
    int getA() const { return 2; }
    int getB() const { return getA(); }
    int getC()       { return 3; }
};

Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;
```

- `foo.getA() == 1`, `foo.getB() == 2`, `foo.getC() == 3`
- `cfoo.getA() == 2`, `cfoo.getB() == 2`, while `cfoo.getC()` gives an error

Ref qualifiers

- member functions can have ref-qualifiers to choose between normal and rvalue reference semantics by appending & or &&

```cpp
struct Bar {
    std::string s;
    Bar(std::string t = "anonymous") : s{t} {} // constructor

    void foo() &  { std::cout << s << " normal instance.\n"; }
    void foo() && { std::cout << s << " temporary instance.\n"; }
};

Bar b{"Fredbob"};
b.foo();     // prints "Fredbob normal instance"
Bar{}.foo(); // prints "anonymous temporary instance"
```

- if you use ref-qualifiers for a member function, all of its overloads need to have ref-qualifiers (cannot be mixed with normal overloads)

Static class members

- static class members (variables or functions) behave like static namespace elements
- static member variables exist only once for all class objects (instead of one copy per object for non-static data)
- static member functions are called without a class object, they can only access static member variables of that class

- static member variables must be defined elsewhere outside the class
- static members can be accessed without further qualification from non-static member functions

Static class members: example

```cpp
class Date {
    int d, m, y;
    static Date defaultDate;
public:
    void setDate(int dd = 0, int mm = 0, int yy = 0) {
        d = dd ? dd : defaultDate.d;
        m = mm ? mm : defaultDate.m;
        y = yy ? yy : defaultDate.y;
    }
    // ...
    static void setDefault(int dd, int mm, int yy) {
        defaultDate = {d, m, y};
    }
};

Date Date::defaultDate{16, 12, 1770}; // definition of defaultDate

int main() {
    Date d;
    d.setDate(); // set to 16, 12, 1770
    Date::setDefault(1, 1, 1900);
    d.setDate(); // set to 1, 1, 1900
}
```

Contents

Initializing and destroying a class object

- when class objects are initialized, a constructor is called
- a destructor is called when the class object is destroyed (i.e. when its lifetime ends)

- a constructor is a special member function of the class
  - the function name is the same as the class name
  - there is no return type and no qualifiers (e.g. const)
  - there can be an argument list (which is used for overload resolution)
  - a constructor without arguments is a default constructor

- the destructor is a special member function of the class
  - the function name is `~class_name`
  - there is no return type and no qualifiers (e.g. const)
  - there are no arguments

Constructors

- constructors consist of two parts:
  - an initializer list (which is executed first)
  - a regular function body (executed second)

- the initializer list specifies how member variables are initialized before the body is executed
  - members should be initialized in the order of their definition
  - if not in the initializer list, a member will be default initialized
  - `const` member variables can only be initialized in the initializer list, not in the constructor body (alternatively: direct initialization in the class declaration)
  - another constructor may be called here (delegated constructor)

- sometimes, constructors are automatically defined by the compiler (see later)

Constructor: examples

```
struct A {
    A() { std::cout << "Hello\n"; } // default constructor
};

struct B {
    int a;
    A b;
    // default constructor implicitly defined by compiler
    // does nothing for a, calls default constructor for b
};

struct Foo {
    int a{123}; float b; const char c;

    // default constructor with initializer list for b, c
    Foo() : b{2.5}, c{7} {} // a has default initializer

    // initializes a, b, c to the given values
    Foo(int av, float bv, char cv) : a{av}, b{bv}, c{cv} {}

    // delegate default constructor, then execute function body
    Foo(float f) : Foo() { b *= f; }
};
```

Calling constructors

- when a class object is initialized, an appropriate constructor is selected by overload resolution
- arguments in the initialization are passed to the constructor
- simplified syntax: `class_type identifier(arguments);` or `class_type identifier{arguments};`
- there are several types of initialization that are very similar, but unfortunately have subtle differences
  - default initialization: `Foo f;`
  - value initialization: `Foo f{};` and `Foo();`
    - beware of most vexing parse for `Foo();`
    - rather use uniform initialization `Foo{};`
  - direct initialization: `Foo f(1, 2, 3);`
  - list initialization: `Foo f{1, 2, 3};`
  - copy initialization: `Foo f = g;`
  - see the reference for the gory details

Converting and explicit constructors

- constructors with exactly one argument are special: they are used for implicit and explicit conversions
- if you do not want implicit conversion, mark the constructor as `explicit`
- in general, it is good practice to always use `explicit`

```cpp
struct Foo {
    Foo(int i);
};

struct Bar {
    explicit Bar(int i);
};

void printFoo(Foo f) { /* ... */ }
void printBar(Bar b) { /* ... */ }

printFoo(123); // implicit conversion, calls Foo::Foo(int)
printBar(123); // ERROR: no implicit conversion
static_cast<Foo>(123); // explicit conversion, calls Foo::Foo(int)
static_cast<Bar>(123); // OK explicit conversion, calls Bar::Bar(int)
```

Copy constructors

- constructors of a class C that have a single argument of type `C&` or `const C&` (preferred) are called copy constructors
- copy constructors are called implicitly by the compiler when an object has to be copied
- the copy constructor is sometimes implicitly defined by the compiler (see later)

```
struct Foo {
    Foo(const Foo& other) { /* ... */ } // copy constructor
};

void doStuff(Foo f) { /* ... */ }

int main() {
    Foo f;
    Foo g(f);   // call copy constructor explicitly
    doStuff(g); // call copy constructor implicitly
}
```

Destructors

- destructors are called when the lifetime of an object ends
  - 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
  - calling the destructor twice on the same object is undefined

```cpp
void f() {
    Foo a;
    {
        Bar b;
        // b.~Bar() is called
    }
    // a.~Foo() is called
}
```

Destructor details

- the destructor is a regular function that can contain any code
    - usually it is used to free resources (see later)
- destructors of member variables are called automatically at the end in reverse order

```cpp
struct Foo {
    Bar a;

    ~Foo() {
        std::cout << "I am being destroyed.\n";
        // a.~Bar() is called
    }
};
```

Contents

Operator overloading

- classes can overload most built-in operators
  - this can be done via special member functions
  - in many cases it can also be done as non-member functions
- general syntax: `return_type operator op (arguments)`
- overloaded operators are selected via regular overload resolution
- overloaded operators are not required to have meaningful semantics (although it is strongly advised for code clarity!)
- almost all operators can be overloaded
  - this includes assignment `=`, call `()`, dereference `*`, address-of `&`, comma `,`
  - exceptions are: scope resolution `::`, member access `.`, member pointer access `.*`, ternary operator `?:`

Unary arithmetic operators

```cpp
struct Int {
    int i;
    Int operator+() const { return *this; }
    Int operator-() const { return Int{-i}; }
};

Int a{123};
+a;  // equivalent to: a.operator+();
-a;  // equivalent to: a.operator-();
```

Binary arithmetic and relational operators

- the expression `lhs op rhs` is (mostly) equivalent to
  - `lhs.operator op(rhs)` or
  - `operator op (lhs, rhs)`
- calls to overloaded operators are treated like regular function calls, meaning the overloaded versions of `||` and `&&` lose their special behavior! (short-circuiting)
- relational operators usually take arguments by const reference

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

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

a + b;   // equivalent to: a.operator+(b);
a == b;  // equivalent to: operator==(a, b);
```

Increment and decrement operators

- overloaded prefix and postfix increment/decrement operators are distinguished by an unused `int` argument
  - `C& operator++(); C& operator--();` overloads the prefix increment/decrement operator, usually modifies the object and then returns `*this`
  - `C operator++(int); C operator--(int);` overloads the postfix increment/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 not overloaded
Int b = a--; // b.i is 124, a.i is 123
--b;     // ERROR: pre-decrement not overloaded
```

Assignment operators

- the simple assignment operator is usually used together with the copy constructor and should have the same semantics (see later)
- all assignment operators usually return `*this`

```
struct Int {
    int i;
    Int& operator=(const Int& other) { i = other.i; return *this; }
    Int& operator+=(const Int& other) { i += other.i; return *this; }
};

Int a{123};
a = Int{456};  // a.i is now 456
a += Int{1};   // a.i is not 457
```

Conversion operators

- constructors can be used to convert values of other types to the class
- similarly, conversion operators can be used to convert class objects to other types
- syntax: `operator type ()`
  - conversion operators have implicit return type `type`
  - they are usually declared `const`
  - `explicit` can be used to prevent implicit conversions
  - `operator bool()` is usually overloaded to be able to use objects in an `if` statement

Conversion operators: example

```cpp
struct Int {
    int i;
    operator int() const { return i; }
};

Int a{123};
int x = a; // OK, x == 123
```

```cpp
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, explicit conversion
```

Argument-dependent lookup

- overloaded operators are usually defined in the same namespace as the type of one of their arguments
- thanks to argument-dependent lookup (ADL), unqualified names of functions are also looked up in the namespaces of all arguments, allowing the following example to compile:

```
namespace A {
    class X {};
    X operator+(const X&, const X&) { return X{}; }
}

int main() {
    A::X x, y;
    A::operator+(x, y);
    operator+(x, y); // ADL uses operator+ from namespace A
    x + y; // ADL finds A::operator+()
}
```

Contents

Derived classes

- any class type (`struct` or `class`) can be derived from one or several base classes
- base classes can themselves be derived from their own base classes, forming an inheritance hierarchy
- general syntax:

```
class class_name : base_specifier_list {
    member_specification
};
```

```
struct class_name : base_specifier_list {
    member_specification
};
```

- we will not cover multiple inheritance in this course, it leads to many problems and should be avoided whenever possible
    - in most cases its use marks a design failure anyway

Derived classes: base specifier list

- the base_specifier_list is a comma-separated list of one or more base_specifiers with the syntax:

```
access_specifier virtual_specifier base_class_name
```

- access_specifier controls the inheritance mode (details see later)
  - it is optional, and one of private, protected, or public
  - public is the most used
  - if not specified, class defaults to private and struct to public
- virtual_specifier is the keyword virtual, it is optional and only used for multiple inheritance
- base_class_name is mandatory and must specify the name of the class to derive from

Derived classes: example

```cpp
class Base {
    int a;
};

class Derived0 : public Base {
    int b;
};

class Derived1 : private Base {
    int c;
};

// multiple inheritance: avoid if possible
class Derived2 : public virtual Base, private Derived1 {
    int d;
};
```

Constructors in derived classes

- constructors of derived classes account for inheritance
  - the direct (non-virtual) base classes are initialized in left-to-right order as they appear in `base_specifier_list`
  - the non-static data members are initialized in order of declaration in the class definition
  - the body of the constructor is executed
  - note: the initialization order is independent of any order in the constructor's member initializer list!
- base classes are default-initialized unless specified otherwise
  - another constructor can be explicitly invoked using the delegated constructor syntax

Constructors in derived classes: example

```
struct Base {
    int a;

    Base() : a{42} { std::cout << "Base::Base()\n"; }
    explicit Base(int av) : a{av} {
        std::cout << "Base::Base(int)\n";
    }
};

struct Derived : Base {
    int b;

    Derived() : b{42} { std::cout << "Derived::Derived()\n"; }
    Derived(int av, int bv) : Base(av), b{bv} {
        std::cout <<  "Derived::Derived(int, int)\n";
    }
};
```

Constructors in derived classes: example (cont.)

Using the classes:

```
int main() {
    Derived derived0;
    Derived derived1(123, 456);
}
```

yields the following output:

```
Base::Base()
Derived::Derived()
Base::Base(int)
Derived::Derived(int, int)
```

Destructors in derived classes

- similar to constructors, destructors of derived classes account for inheritance as well
    - the body of the destructor is executed
    - the destructors of all non-static members are called in reverse order of declaration
    - the destructors of all (non-virtual) base classes are called in reverse order of construction

- the order in which 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 `base_specifier_list`

Destructors in derived classes: example

```
struct Base0 {
    ~Base0() { std::cout << "Base0::~Base0()\n"; }
};

struct Base1 {
    ~Base1() { std::cout << "Base1::~Base1()\n"; }
};

struct Derived : Base0, Base1 {
    ~Derived() { std::cout << "Derived::~Derived()\n"; }
};

int main() {
    Derived derived;
}
```

yields the output

```
Derived::~Derived()
Base1::~Base1()
Base0::~Base0()
```

Qualified and unqualified name lookup

- like in namespaces and scopes, you can hide names in an inheritance hierarchy (but you should not do that!)
- in general, declarations in derived classes hide declarations in base classes
- use qualified name lookup (with scope resolution operator `::`) to resolve ambiguities

```
struct A {
    void a();
};

struct B : A {
    void a();
};

int main {
    B b;
    b.a();    // calls B::a()
    b.A::a(); // calls A::a()
}
```

Inheritance modes

- the inheritance mode in the `access_specifier` specifies the access to the base class:
  - `public` means the public base class members are public, and the protected members are protected
  - `protected` means the public and protected base class members are only accessible for class members / friends of the derived class and its derived classes
  - `private` means the public and protected base class members are only accessible for class members / friends of the derived class

Inheritance modes (continued)

- `public` inheritance also models a subtyping (IS-A) relationship
  - this is the inheritance mode used in most cases
  - references and pointers to a base object will also accept a derived object
  - a derived class must maintain the same class invariants (usually enforced via constructors) as its base classes
  - a derived class must not strengthen pre-conditions or weaken post-conditions of any member function
- `private` and `protected` inheritance modes are rarely used

Contents

Members in derived classes

- inheritance in C++ is by default non-polymorphic
- members in derived classes can hide members in the base class
- the type of the object determines which member is referred to

```cpp
struct Base {
    void foo() { std::cout << "Base::foo()\n"; }
};

struct Derived : Base {
    void foo() { std::cout << "Derived::foo()\n"; }
};

int main() {
    Derived d;
    Base& b = d; // works, Derived IS-A Base

    d.foo(); // prints Derived::foo()
    b.foo(); // prints Base::foo()
}
```

Virtual functions

- non-static member functions can be marked `virtual`
  - enables dynamic dispatch for this function
  - allows the function to be overridden in derived classes
  - a class with at least one virtual function is polymorphic

- only the function in the base class needs to be marked `virtual`, the overriding functions in derived classes do not need to be (and should not be) marked `virtual`
  - instead they are marked `override`

- when calling an overridden virtual function through a reference (or pointer) to a base object, the implementation of the derived class will be invoked
  - this can be suppressed by using qualified name lookup

Virtual functions: example

```cpp
struct Base {
    virtual void foo() { std::cout << "Base::foo()\n"; }
};

struct Derived : Base {
    void foo() override { std::cout << "Derived::foo()\n"; }
};

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()
}
```

Overriding virtual functions

- a function overrides a virtual base class function, if:
  - the name, the parameters and the qualifiers are the same
  - the return type is the same or covariant (see later)
- if these conditions are not met, the function hides the virtual base class function!
  - to avoid surprises, use the `override` specifier after the declarator, allowing the compiler to help you

- a function overriding a virtual base class function is also automatically virtual, and can be overridden by further-derived classes
- a base class function does not need to be visible to be overridden

Overriding virtual functions: example

Specifying override helps the compiler catch mistakes:

```cpp
struct Base {
    virtual void foo(int i);
    virtual void bar();
};

struct Derived : Base {
    void foo(float i) override; // ERROR: does not override foo(int)
    void bar() const override;  // ERROR: does not override bar()
};
```

Overriding virtual functions: example (cont.)

Not using `override` can lead to surprises:

```cpp
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 only finds Derived::foo(int)
    b.foo(); // invokes Base::foo()
}
```

More overriding examples

The actual type of the object determines which override is called:

```cpp
struct A {
    virtual void foo();
    virtual void bar();
    virtual void baz();
};

struct B : A {
    void foo() override;
    void bar() override;
};

struct C : B {
    void foo() override;
};
```

```cpp
int main() {
    C c;
    A& cr = c;

    cr.foo(); // invokes C::foo()
    cr.bar(); // invokes B::bar()
    cr.baz(); // invokes A::baz()

    B b;
    A& br = b;

    br.foo(); // invokes B::foo()
    br.bar(); // invokes B::bar()
    br.baz(); // invokes A::baz()
}
```

Covariant return types

- virtual and overriding functions can have covariant return types
    - both types must be single-level references (or pointers) to classes
    - the referenced (or pointed-to) class in the base class function must be a direct or indirect base class of the referenced (or pointed-to) class in the derived class function
    - the return type in the derived class must be as const-qualified as the return type in the base class
- usually, the referenced (or pointed-to) class in the derived class function is the derived class itself

Covariant return types: example

```
struct Base {
    virtual Base& foo();
    virtual Base* bar();
};

struct Derived : Base {
    Derived& foo() override; // overrides Base::foo()
    int bar() override; // ERROR: overrides Base::bar() but
                        // has non-covariant return type
};
```

Marking functions/classes as final

- to prevent overriding a function it can be marked as final:

```cpp
struct Base {
    virtual void foo() final;
};

struct Derived : Base {
    void foo() override; // ERROR: overriding final function
};
```

- you can also mark as class as final to prevent inheritance:

```cpp
struct Base final {
    virtual void foo();
};

struct Derived : Base { // ERROR: deriving final class
    void foo() override;
};
```

Contents

Constructors and virtual functions

Caution when using virtual functions during construction or destruction!

- during construction, virtual functions behave like regular functions:

```
struct Base {
    Base() { foo(); }
    virtual void foo();
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Derived d; // on construction, Base:foo() is called!
}
```

Virtual destructors

- derived objects can be deleted through a pointer to the base class
  - this leads to undefined behavior unless the destructor in the base class is `virtual`

- hence, the destructor in a base class should either be
  - protected and non-virtual or
  - public and virtual (this should be the default)

```cpp
struct Base {
    virtual ~Base() {}
};

struct Derived : Base { };

int main() {
    Base* b = new Derived();
    delete b; // OK thanks to virtual destructor
    // but in general: don't use naked pointers/new/delete!
}
```

Slicing

- inheritance hierarchies need to be handled via references (or pointers)!
    - otherwise you might get “surprises”
    - and even then, care has to be taken!

- simple slicing example:

```cpp
struct A {
    int x;
};

struct B : A {
    int y;
};

void f() {
    B b;
    A a = b; // a will only contain x, y is lost
}
```

Slicing: example

```
struct A {
    int x;
};

struct B : A {
    int y;
};

void g() {
    B b1;
    B b2;

    A& ar = b2;
    ar = b1;    // b2 now contains a mixture of b1 and b2!!!
}
```

- it is usually a good idea to disallow copying of objects in class hierarchies (e.g. by `= delete`'ing the copy constructor and assignment operator, see later)
  - a Cloneable pattern can be used (see later)

Converting references/pointer in an inheritance hierarchy

- to convert references (or pointers) in an inheritance hierarchy in a safe manner, you can use

```
            dynamic_cast< new_type > ( expression )
```

  - `new_type` is a reference (or pointer) 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
  - runtime check if `new_type` is a base of the actual polymorphic type of `expression`
  - if the check fails, throws an exception for reference types (or returns `nullptr` for pointer types)
  - this requires run-time type information, which has overhead!

dynamic_cast example

```
struct A {
    virtual ~A() = 0;
};

struct B : A {
    void foo() const;
};

struct C : A {
    void bar() const;
};

// bad design, just an illustration!
void baz(const A* aptr) {
    if (auto bptr = dynamic_cast<const B*>(aptr)) {
        bptr->foo();
    }
    else if (auto cptr = dynamic_cast<const C*>(aptr)) {
        cptr->bar();
    }
}
```

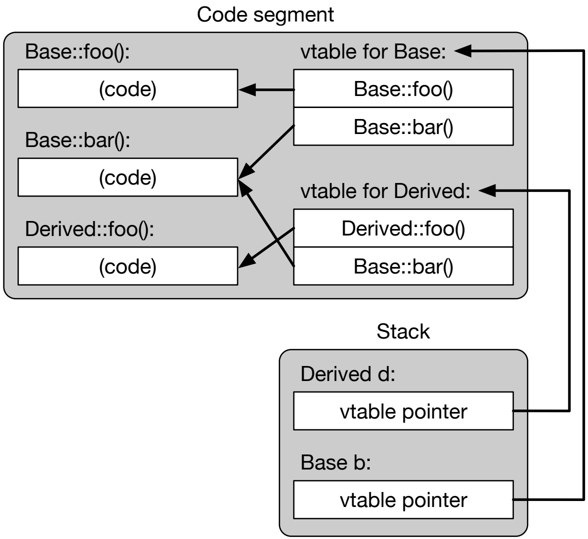
Implementing virtual functions: vtables

- polymorphism does not come for free, it has overhead!
  - dynamic dispatch has to be implemented somehow
  - the C++ standard does not prescribe a specific implementation
  - most compiler use vtables to resolve virtual function calls

- vtables work like this:
  - 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 example

```cpp
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();
}
```



Cost of virtual functions

- run-time cost:
  - each virtual function call has to: follow the pointer to the vtable, follow the pointer to the actual function
  - these steps might lead to cache misses
  - can be very noticeable when invoking a virtual function millions of times
- memory cost:
  - polymorphic objects have larger size, as it has to store a pointer to the vtable (and the vtable itself)
  - in the previous example, both `Base` and `Derived` occupy 8 bytes of memory despite having no data members

Contents

Pure virtual functions and abstract classes

- a virtual function can be marked as a pure virtual function by adding `= 0` at the end of the declarator/specifiers
- any class declaring or inheriting at least one pure virtual function is an abstract class
- abstract classes are special:
  - they cannot be instantiated
  - but they can be used as a base class (defining an interface)
  - references (and pointers) to abstract classes can be declared
  - caution: calling a pure virtual function in the constructor or destructor of an abstract class is undefined behavior!

Abstract class: examples

```cpp
struct Base {
    virtual void foo() = 0;
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Base b;        // ERROR: abstract class
    Derived d;
    Base& dr = d;
    dr.foo();      // calls Derived::foo()
}
```

Abstract class: examples (cont.)

An out-of-line definition of a pure virtual function can be provided:

```cpp
struct Base {
    virtual void foo() = 0;
};

void Base::foo() { /* do stuff */ }

struct Derived : Base {
    void foo() override { Base::foo(); }
};
```

Abstract class: examples (cont.)

The destructor can be marked as pure virtual

- useful when 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: Base is abstract
}
```

Abstract class: examples (cont.)

- abstract class cannot be instantiated, but can be referred to via references or pointers

```cpp
struct Base {
    virtual ~Base() {}
    virtual void foo() const = 0;
};

struct Derived : Base {
    void foo() const 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!
```

Contents

The other class type: enum classes

- we can define enumerations, for example to define descriptive names instead of using "magic" values

```cpp
enum class TrafficLight {
    red, yellow, green
};
```

  - by default, the underlying type is an `int`
  - by default, enumerator values are assigned increasing from 0

- access names from enum via scope resolution operator
- there are no implicit conversions to integers

```cpp
TrafficLight tl = TrafficLight::yellow;
tl = 0; // ERROR: 0 is not a TrafficLight
tl = static_cast<TrafficLight>(0); // OK, explicit conversion
```

- see reference for more details

Contents

RAII - Resource Acquisition is Initialization

One of the most important and powerful concepts of C++ is

RAII - Resource Acquisition is Initialization

- bind the lifetime of a resource (e.g. memory, sockets, files, mutexes, database connections) to the lifetime of an object
- guarantees availability of the resource during the lifetime of the object
- guarantees that the resource is released when the lifetime of the object ends
- objects should have automatic storage duration (so its lifetime is managed automatically)

RAII - how to implement it

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 (see later)

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

RAII: typical usage example

Write a string to a file, in thread- and exception-safe manner:

```cpp
void writeMessage(std::string message) {
    // mutex to protect file access across threads
    static std::mutex myMutex;
    // lock mutex before accessing file
    std::lockguard<std::mutex> lock(myMutex);

    std::ofstream file("message.txt"); // try to open file
    if (!file.is_open()) // throw exception in case it failed
        throw std::runtime_error("unable to open file");

    file << message << "\n";

    // no cleanup needed
    // - file will be closed when leaving scope (regardless of
    //   exception) by ofstream destructor
    // - mutex will be unlocked when leaving scope (regardless of
    //   exception) by lockguard destructor
}
```

Contents

Copy semantics

- construction and assignment of classes employs copy semantics in most cases
  - by default, a shallow copy is created
  - usually not particularly relevant for built-in types
  - very relevant for user-defined types
- what to consider when copying:
  - copying may be expensive
  - copying may be unnecessary or even unwanted (see slicing example earlier)
  - an object being copied to might manage resources (RAII)

Copy constructor

- the copy constructor is invoked whenever an object is initialized from an object of the same type, its syntax is

```
            class_name ( const class_name& )
```

- for class type `T` and objects `a`, `b`, the copy constructor is invoked for:
  - copy initialization: `T a = b;`
  - direct initialization: `T a{b};`
  - function argument passing: `f(a);` where `void f(T t)`
  - function return: `return a;` inside a function `T f();` (if `T` has no move constructor, see later)

Copy constructor: example

```
class A {
    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

- the copy assignment is invoked when an object appears on the left-hand side of an assignment with an lvalue on the right-hand side
    - two options:

```
class_name& operator= ( const class_name& )
```

```
class_name& operator=( class_name )
```

    - typically the first option is preferred (except for the copy-and-swap idiom later)

- copy assignment is called whenever it is selected by overload resolution
- returns a reference to the object itself (i.e. `*this`) to allow chaining of assignments

Copy assignment: example

```
class A {
    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
}
```

Copy constructor: implicit declaration

- 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 `= delete` 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 reference for details...

Copy assignment: implicit declaration

- 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 may or may not be **defined**!

- the implicitly declared copy assignment operator is **defined** as `= delete` 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 reference for details...

Copy constructor and assignment: implicit definition

- if it is not deleted, the compiler defines the implicitly-declared copy constructor
    - only if it is actually used
    - it will perform 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 compile defines the implicitly-declared copy assignment operator
    - only if it is actually used
    - it will perform 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

Implicit copy construction/assignment: example

```
struct A {
    const int v;

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

int main() {
    A a1{42};

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

Implementing custom copy operations

- 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
- guidelines for implementing custom copy operations:
  - you should provide either provide both copy constructor **and** copy assignment, or neither of them
  - 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: example

```cpp
class A {
    unsigned capacity;
    std::unique_ptr<int[]> memory; // better would be: std::vector<int>
public:
    explicit A(unsigned cap) : capacity{cap},
         memory{std::make_unique<int[]>(capacity)} {}

    A(const A& other) : A(other.capacity) {
        std::copy(other.memory.get(),
                  other.memory.get() + other.capacity, memory.get());
    }

    A& operator=(const A& other) {
        if (this == &other) // check for self-assignment
            return *this;
        if (capacity != other.capacity) { // attempt resource reuse
            capacity = other.capacity;
            memory = std::make_unique<int[]>(capacity);
        }
        std::copy(other.memory.get(),
                  other.memory.get() + other.capacity, memory.get());
        return *this;
    }
```

The Rule of Three

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 destructor usually implies some custom cleanup logic, which also needs to be executed by copy assignment (and vice versa)
  - having a user-defined copy constructor usually implies some custom setup logic, which also 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. raw memory)

Contents

Move semantics

- copy semantics are sometimes not wanted or have unnecessary overhead
  - an object might not want to share/copy a resource it is managing
  - an object might be immediately destroyed after it is copied
- move semantics provide a solution
  - move constructors and move assignment operators typically “steal” the resources of the argument
  - leave the argument in valid but indeterminate state
  - greatly enhances performance in some cases

Move constructor

- the move constructor is invoked when an object is initialized from an rvalue reference of the same type

```
class_name ( class_name&& ) noexcept
```

  - the `noexcept` keyword is optional, but should be added to indicate that the move constructor never throws an exception

- overload resolution determines if copy or move constructor of the object should be called
- the function `std::move` from `#include <utility>` can be used to convert an lvalue to an rvalue reference
- the argument being moved from does not need its resources anymore, so we can simply steal them

Move constructor: example

```
struct A {
    A();
    A(const A& other);     // copy constructor
    A(A&& other) noexcept; // move constructor
};

int main() {
    A a1;
    A a2(a1);             // calls copy constructor
    A a3(std::move(a1)); // calls move constructor
}
```

- the move constructor for class type `T` and objects `a,b` is invoked for
  - direct initialization: `T a{std::move(b)};`
  - copy initialization: `T a = std::move(b);`
  - passing arguments to a function `void f(T t);` via `f(std::move(a))`
  - returning from a function `T f();` via `return a;`

Move assignment

- the move assignment is invoked when an object appears on the left-hand side of an assignment with an rvalue reference on the right-hand side

```
            class_name& operator=( class_name&& ) noexcept
```

  - the noexcept keyword is optional, but should be added to indicate that the move assignment never throws an exception

- overload resolution determines if the copy or move assignment operator is selected
- the function `std::move` from `#include <utility>` can be used to convert an lvalue to an rvalue reference
- the argument being moved from does not need its resources anymore, so we can simply steal them
- as with other assignments, move assignment returns a reference to the object itself (i.e. `*this`) to allow chaining

Move assignment: example

```
struct A {
    A();
    A(const A&);      // copy constructor
    A(A&&) noexcept; // move constructor

    A& operator=(const A&);      // copy assignment
    A& operator=(A&&) noexcept; // move assignment
};

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
}
```

Move constructor: implicit declaration

- the compiler will implicitly declare a `public` move constructor, if all of 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 `= delete` 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 reference for details...

Move assignment: implicit declaration

- the compiler will implicitly declare a `public` move assignment operator if all of 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 copy assignment operator is defined as = `delete` 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 reference for details...

Move constructor and assignment: implicit definition

- if it is not deleted, the compiler defines the implicitly-declared move constructor
    - only if it is actually used
    - it will perform 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
    - it will perform 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

Implicit move construction/assignment: example

```
struct A {
    const int v;

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

int main() {
    A a1{42};

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

Implementing custom move operations

- 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 implement custom move operations:
    - you should either provide both move constructor **and** move assignment, or neither of them
    - the move assignment operator should usually include a check to detect self-assignment
    - the move operations typically do not allocate new resources, but steal the resources from the argument
    - the move operations should leave the argument in a valid (but indeterminate) state
    - any previously held resources must be cleaned up properly!

Implement custom move operations: example

```cpp
class A {
    unsigned capacity;
    std::unique_ptr<int[]> memory; // better would be: std::vector<int>

public:
    explicit A(unsigned cap) : capacity{cap},
        memory{std::make_unique<int[]>(capacity)} {}

    A(A&& other) noexcept : capacity{other.capacity},
        memory{std::move(other.memory)}  {
        other.capacity = 0;
    }

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

        capacity = other.capacity;
        memory = std::move(other.memory);

        other.capacity = 0;
        return *this;
    }
```

The Rule of Five

The Rule of Five

- if a class wants move semantics, it has to define all five special member functions
- if a class wants only move semantics, it still has to define all five special member functions, but define the copy operations as `= delete`

- explanation:
  - if a class follows the Rule of Three, the move operations are defined as deleted
  - not adhering to the Rule of Five usually does not lead to incorrect code, but many optimization opportunities might be inaccessible, if no move operations are defined

The Rule of Zero

The Rule of Zero

- classes not dealing with ownership (e.g. of resources) should not have custom destructors, copy/move constructors or copy/move assignment operators
- classes that do deal with ownership should do so exclusively (and follow the Rule of Five)

Special case: polymorphic base classes

- polymorphic base classes should have a public virtual destructor
- in most of the cases you should then = delete all copy/move operations to prevent slicing:

```cpp
class BaseOfFive {
public:
    virtual ~BaseOfFive() = default;
    BaseOfFive(const BaseOfFive&) = delete;
    BaseOfFive(BaseOfFive&&) = delete;
    BaseOfFive& operator=(const BaseOfFive&) = delete;
    BaseOfFive& operator=(BaseOfFive&&) = delete;
};
```

- in many cases you may then want to provide polymorphic clone functionality

Polymorphic cloning

- to enable copying of polymorphic objects, you can define a factory method `clone()`:

```
struct A {
    A() {} // default constructor
    virtual ~A() = default; // base class with virtual destructor
    A(const A&) = delete;   // prevent slicing
    A(A&&) = delete;
    A& operator=(const A&) = delete;
    A& operator=(A&&) = delete;

    virtual std::unique_ptr<A> clone () { return /* A object */; }
};

struct B : A {
    std::unique_ptr<A> clone() override { return /* B object */; }
};

auto b = std::make_unique<B>();
auto a = b->clone(); // cloned copy of b
```

- we can use CRTP for polymorphic cloning, see later

Copy and swap

- if copy assignment cannot benefit from resource reuse, the copy and swap idiom may be useful:
  - the class only defines `class_name& operator=( class_name )` copy-and-swap assignment operator
  - it acts as both copy and move assignment operator, depending on the value category of the argument
- implementation:
  - exchange the resource between the argument and `*this`
  - let the destructor clean up the resources of the argument
- caution: while convenient, this might not be the fastest approach

Copy and swap: example

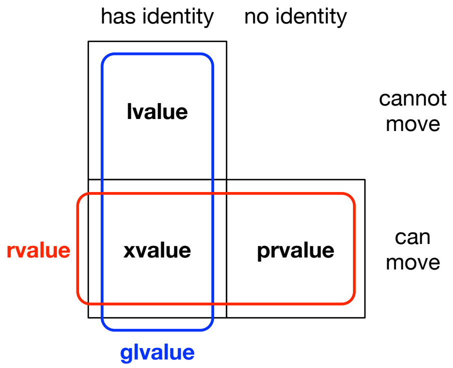
```cpp
class A {
    unsigned capacity;
    std::unique_ptr<int[]> memory; // better would be: std::vector<int>

public:
    explicit A(unsigned cap) : capacity{cap},
        memory{std::make_unique<int[]>(capacity)} {}

    A(const A& other) : A(other.capacity) {
        std::copy(other.memory.get(),
                  other.memory.get() + other.capacity, memory.get());
    }

    A& operator=(A other) { // copy/move constructor will create 'other'
        std::swap(capacity, other.capacity);
        std::swap(memory, other.memory);

        return *this;
    } // destructor of 'other' cleans up resources formerly held by *this
};
```

Contents

Value categories

We need to be more specific about lvalues and rvalues:

- 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 elision

Compilers have to omit copy/move construction of class objects under certain circumstances:

- in a `return` statement, when the operand is a prvalue of the same class type as the function return type:

```cpp
T f() {
    return T();
}

f(); // only one call to default constructor of T
```

- in the initialization of an object, when the initializer expression is a prvalue of the same class type as the variable type:

```cpp
T x = T{T{f()}}; // only one call to default constructor of T
```

Copy elision (cont.)

- these optimizations are required from the compiler
- this is even the case if the copy/move constructor or destructor would have side effects!
- for more details, see the reference

- other optimizations (such as NRVO - named return value optimization) are allowed but optional for the compilers
- hence code that relies on side-effects of copy/move constructors and destructors is not portable!

Copy elision: example

```cpp
#include <iostream>

struct A {
    int a;

    A(int aa) : a{aa} {
        std::cout << "constructed\n";
    }

    A(const A& other) : a{other.a} {
        std::cout << "copy-constructed\n";
    }
};

A foo() {
    return A{42};
}

int main() {
    A a = foo(); // prints only "constructed"
}
```

Contents

Ownership semantics

- RAII and move semantics enable ownership semantics
    - a resource should be owned (encapsulated) by exactly one object at all times
    - ownership can only be transferred explicitly by moving the respective object

- always use ownership semantics when managing resources in C++ (such as memory, file handles, sockets etc.)!

Contents

Owning smart pointer: `std::unique_ptr`

- `std::unique_ptr` is a smart pointer implementing ownership semantics for arbitrary pointers
  - assumes unique ownership of another object through a pointer
  - automatically disposes of that object when `std::unique_ptr` goes out of scope
  - a `std::unique_ptr` can be used (almost) exactly like a raw pointer, but it can only be moved, not copied
  - a `std::unique_ptr` may be empty (i.e. owning no object)

- `std::unique_ptr` is defined in `#include <memory>`
  - it is a template class (see later), and can be used for any type
  - use `std::unique_ptr< type >` instead of the raw pointer type `type*`

- ALWAYS use `std::unique_ptr` over raw pointers!

std::unique_ptr usage

- creation:
  - `std::make_unique<type>(arg0, ..., argN)`, where `arg0, ..., argN` are passed to the constructor of `type`
- dereferencing, subscript, member access:
  - dereferencing `*`, subscripting `[]`, and member access `->` operators 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 (like 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 (very rarely required)

std::unique_ptr example

```
struct A {
    int a;
    int b;

    A(int aa, int bb) : a{aa}, b{bb} {}
};

void foo(std::unique_ptr<A> aptr) { // assume ownership
    /* do something */
}

void bar(const A& a) { // does not assume ownership
    /* do something */
}

int main() {
    auto aptr = std::make_unique<A>(42, 123);
    int a = aptr->a;
    bar(*aptr);             // retain ownership
    foo(std::move(aptr)); // transfer ownership
}
```

std::unique_ptr array example

```cpp
std::unique_ptr<int[]> foo(unsigned size) {
    auto buffer = std::make_unique<int[]>(size);

    for (unsigned i = 0; i < size; ++i)
        buffer[i] = i;

    return buffer; // transfers ownership to caller
}

int main() {
    auto buffer = foo(42);

    /* do stuff with buffer */
}
```

- although here using a `std::vector` would be better...

Shared smart pointer: `std::shared_ptr`

- `std::shared_ptr` is a smart pointer implementing shared ownership
  - a resource can be simultaneously shared with several owners
  - the resource will only be released once the last owner releases it
  - multiple `std::shared_ptr` objects can own the same raw pointer, implemented through reference counting
  - a `std::shared_ptr` can be copied and moved

- `std::shared_ptr` is defined in `#include <memory>`
  - it is a template class, and can be used for any type
  - use `std::make_shared` for creation, otherwise analogous to `std::unique_ptr`

- `std::shared_ptr` has overhead (reference counting) and should only be used when explicitly necessary
- to break cycles, you can use `std::weak_ptr`, see reference

std::shared_ptr example

```cpp
#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

- `std::unique_ptr` represents ownership
  - used for dynamically allocated objects; often 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 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 (usually `std::unique_ptr`)
  - very, very rarely, raw pointers are desired as function parameters or return types
    - if ownership is not transferred, but there might be no object (i.e. `nullptr`) $\Rightarrow$ consider using `std::optional` instead (see later)
    - if ownership is not transferred, but pointer arithmetic is required

Usage guidelines: references

- references grant temporary access to an object without assuming ownership
  - a reference can be obtained from a smart pointer through the dereferencing 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 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 reference if ownership is not transferred
  - should be passed by value if ownership is transferred (invoke using `std::move`)

Usage guidelines: example

```cpp
struct A { };

// reads a without assuming ownership
void readA(const A& a);
// may read and modify a but does not assume ownership
void readWriteA(A& a);
// works on a copy of A
void workOnCopyOfA(A a);
// assumes ownership of A
void consumeA(A&& a);

int main() {
    A a;

    readA(a);
    readWrite(a);
    workOnCopyOfA(a);
    consumeA(std::move(a)); // cannot call without std::move
}
```

Some nasty examples why raw pointers are evil

!!! DO NOT DO THIS !!!

- use of uninitialized pointers:

```
int* i;
*i = 5; // OUCH: writing to random memory location
```

- leaking memory:

```
void f() {
    int* i = new int{5};
    // ... use i ...
} // OUCH: memory of i not freed, memory leak!
```

- undefined behavior:

```
int* i = new int{5};
// ... use i ...
delete i;
delete i; // OUCH: freeing already freed pointer, undefined!
```

Some nasty examples why raw pointers are evil (cont.)

!!! DO NOT DO THIS !!!

- not checking for null pointer:

```
void f(int* i) {
    *i = 5; // OUCH: i might be nullptr
}

f(nullptr); // OUCH: this might crash or worse..
```

- unchecked pointer arithmetic:

```
int *i = new int[5] {1, 2, 3, 4, 5};
int *j = i;     // i and j now point to same location

*j = 10;        // ok, i[0] is now 10
*j++ = 11;      // ok, i[0] is now 11, j points to i[1]
*(++j) = 12;    // ok, i[2] is now 12, j points to i[2]
*(j + 10) = 13; // OUCH: overwriting something unrelated to i/j
```

Use ownership semantics and smart pointers!

- make sure to use ownership semantics, i.e. RAII and move semantics
- if tempted to use raw pointers, think again! And again!
    - use smart pointers instead (`std::unique_ptr` or, rarely, `std::shared_ptr`)

Contents

What is generic programming?

- functionality is often independent of a specific type `T`, indeed it is generic
- examples:
  - `std::swap(T& a, T& b)`
  - `std::vector<T>`
  - and many others

- 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 parametrized by one or more template parameters
  - type template parameters
  - non-type template parameters
  - template template parameters
- 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 later), templates are instantiated
  - template instantiation actually compiles the code for the respective specialization

Templates: simple example

Simplified version of `std::vector`:

```
template <class T>    // T is a type template parameter
class vector {
public:
    // ...
    void push_back(T&& element);
    // ...
};

class A;

int main() {
    vector<int> vectorOfInt; // int is substituted for T
    vector<A> vectorOfA;     // A is substituted for T
}
```

Templates: syntax

- template declaration 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

Template parameters: type

- type template parameters are placeholders for arbitrary types
- syntax: `typename name` or `class name` (both variants are equivalent)
- `name` can be omitted, e.g. in forward declarations
- in the body of the template declaration, `name` is a type alias for the type supplied during instantiation

```
template <class, class>
struct Baz;

template <typename T>
struct Foo {
    T bar(T t) {
        return t + 42;
    }
};
```

Template parameters: non-type

- non-type template parameters are placeholders for certain values
- syntax: `type name`
- `name` can be omitted, e.g. in forward declarations
- `type` can 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 <typename T, unsigned N>
class Array {
    T storage[N];

public:
    T& operator[](unsigned i) {
        assert(i < N);
        return storage[i];
    }
};
```

Template parameters: template

- type template parameters can themselves be templated
- syntax: `template < parameter_list > typename name` or `template < parameter_list > class name`
- `name` can 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, unsigned> class ArrayType>
class Foo {
    ArrayType<int, 42> someArray;
};
```

- rarely used, should be avoided where 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 cannot be followed by template parameters without default values

```
template <typename T = std::byte, unsigned Capacity = 1024>
class Buffer {
    T storage[Capacity];
};
```

Using templates

- 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 (see later)

Using templates with type template arguments

- template arguments for type template parameters must name a type (which may be incomplete)

```cpp
template <class T1, class T2 = int, class T3 = double>
class Foo { };

class A;

int main() {
    Foo<int> foo1;
    Foo<A> foo2;
    Foo<A*> foo3;
    Foo<int, A> foo4;
    Foo<int, A, A> foo5;
}
```

Using templates with non-type template arguments

- template arguments for non-type template parameters must be (converted) constant expressions
  - converted constant expressions must be evaluated at compile-time, some limited set of implicit conversions may take place
  - the (possibly implicitly converted) type of the expressions must match the type of the template parameter

```
template <unsigned N>
class Foo { };

int main() {
    Foo<42u> foo1;  // OK: no conversion
    Foo<42>  foo2;  // OK: numeric conversion
}
```

- there are restrictions for non-type template parameters of reference or pointer type:
  - may not refer to a subobject (non-static class member, base subobject), or a temporary object, or a string literal

Using templates with template template parameters

- template arguments for template template arguments must name a class template or template alias

```
#include <array>

template <typename T, unsigned long N>
class MyArray { };

template <template <class, unsigned long> class Array>
class Foo {
    Array<int, 42> bar;
};

int main() {
    Foo<MyArray> foo1;
    Foo<std::array> foo2;
}
```

Example: class template

- example of a class template:

```cpp
template <typename T, unsigned long N>
class MyArray {
    T storage[N];

public:
    // ...

    T& operator[](unsigned long index) {
        assert(index < N);
        return storage[index];
    }

    const T& operator[](unsigned  long index) const {
        assert(index < N);
        return storage[index];
    }

    // ...
};
```

Example: function template

- example of a function template:

```cpp
template <typename T>
void swap(T& a, T& b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}

class A { };

int main() {
    A a1;
    A a2;

    swap<A>(a1, a2);
    swap(a1, a2);     // also OK: template arguments are deduced
}
```

Example: alias template

- example of a alias template:

```
namespace something::extremely::nested {

    template <typename T, typename R>
    class Handle  { };

} // end 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 template

- example of a variable template:

```cpp
template <typename T>
constexpr T pi = T{3.1415926535897932385L};

template <typename T>
T area(T radius) {
    return pi<T> * radius * radius;
}

int main() {
    double a = area<double>(1.0);
}
```

Example: class member template

- example of a class member template:

```cpp
#include <iostream>
#include <array>

struct Foo {
    template <typename T>
    using ArrayType = std::array<T, 42>;

    template <typename T>
    void printSizeOf() {
        std::cout << sizeof(T) << "\n";
    }
};

int main() {
    Foo::ArrayType<int> intArray;

    Foo foo;
    foo.printSizeOf<Foo::ArrayType<int>>();
}
```

Template instantiation

- a function or class template by itself is not a type, an object, or any other entity
    - no code is generated from a file that contains only template definitions!
- a template specialization must be instantiated for any code to appear
    - compiler generates an actual function or class for a template specialization
    - some compiler errors will only turn up at this point!

- there are two kinds of template instantiation
    - 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

- explicitly force instantiation of a template specialization
  - syntax for class templates:
    `template class template_name < argument_list >;` or
    `template struct template_name < argument_list >;`
  - syntax for function templates:
    `template ret_type name < argument_list > (param_list);`

- explicit instantiations have to follow the one definition rule
- generates code for the function specialization or class specialization including all of its member functions
- the definition of the template must be visible at the point of explicit instantiation

Explicit template instantiation: example

```cpp
template <typename T>
struct A {
    T foo(T value) { return value + 42; }

    T bar() { return 42; }
};

template <typename 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

- implicit instantiation is caused by using a template specialization in a context that requires a complete type
    - this only happens if the template specialization has not already been explicitly instantiated
    - members of a class template are only instantiated if they are actually used
- the definition of the template must be visible at the point of implicit instantiation
    - that means for implicit instantiation, the definitions must be provided, usually in a header file!

Implicit template instantiation: example

```cpp
template <typename 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

    // note: no error, even though A::bar is never defined

    A<float>* aptr;     // does not instantiate A<float>!
}
```

Explicit vs. implicit instantiation

Implicit instantiation:

- advantages:
  - template can be used with any suitable type
  - no unnecessary code is generated
- disadvantages:
  - definition has to be provided in header file
  - the user of our templates has to compile them

Explicit instantiation:

- advantages:
  - explicit instantiations can be compiled into a library
  - definitions can be encapsulated in a source file
- disadvantage: limits usability of our templates
  - cannot implicitly instantiate the template for an unforeseen type

Instantiation

- the compiler actually generates code for instantiations
  - conceptually, the template parameters are replaced by the template arguments
  - each instantiation for different template arguments will generate code
  - this can substantially increase code size and compilation times!

- instantiations are generated locally for each compilation unit
  - the same instantiation can exist in different compilation units without violating the one definition rule (ODR)

Inline vs. Out-of-line definition

- out-of-line definitions are generally preferred, even when defining class templates in header files
  - better readability of interface

```
template <typename T>
struct A {
    T value;

    A(T v);

    template <typename R>
    R convert();
};

template <typename T>      // out-of-line definition
A<T>::A(T v) : value{v} {}

template <typename T>
template <typename R>      // out-of-line definition
R A<T>::convert() { return static_cast<R>(value); }
```

Templates and dependent names

- 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 <typename 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
    }
};
```

Templates and dependent names (cont.)

- 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

```cpp
struct A {
    using MemberTypeAlias = float;
};

template <typename T>
struct B {
    using AnotherMemberTypeAlias = typename T::MemberTypeAlias;
};

int main() {
    B<A>::AnotherMemberTypeAlias value = 42.0f; // value is type float
}
```

Templates and dependent names (cont.)

- 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 <typename T>
struct A {
    template <typename R>
    R convert(T value) { return static_cast<R>(value); }
};

template <typename T>
T foo() {
    A<int> a;

    return a.template convert<T>(42);
}
```

Reference collapsing

- templates and type aliases can form references to references:

```
template <typename 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 (so our `x` here is an rvalue reference)
  - any other combination forms an lvalue reference

Contents

Specializing templates explicitly

- while templates are for generic code, sometimes we might want to modify the behavior of templates for specific template arguments
  - for example, a templated search method could employ different algorithms on arrays (binary search) vs. linked lists (linear search)
- to achieve this, templates can be explicitly specialized by defining specific implementations for certain template arguments
  - all template arguments can be specified: full specialization
  - some template arguments specified: partial specialization

Full specialization

- to define a specific implementation for a full set of template arguments:

```
template <> declaration
```

- has to appear after the declaration of the original template

```
template <typename T>
class MyContainer {
    /* generic implementation */
};

template <>
class MyContainer<long> {
    /* specific implementation for long */
};

int main() {
    MyContainer<float> a; // uses generic implementation
    MyContainer<long> b;  // uses specific implementation
}
```

Partial specialization

- to define a specific implementation for a partial set of template arguments:

```
template < parameter_list > class name < argument_list >
```

```
template < parameter_list> struct name < argument_list >
```

- has to appear after the declaration of the original template
- only class templates can be partially specialized (use function overloads for functions instead)

Partial specialization: example

```cpp
template <typename C, typename T>
class SearchAlgorithm {
    void find(const C& container, const T& value) {
        /* do linear search */
    }
};

template <typename T>
class SearchAlgorithm<std::vector<T>, T> {
    void find(const std::vector<T>& container, const T& value) {
        /* do binary search */
    }
};
```

Contents

Template argument deduction

- to instantiate a template, all template arguments have to be known
- some template arguments can be deduced from the context, so you do not have to specify all template arguments all the time

- for example:

```
template <typename 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

- to deduce template arguments in function calls, the types of the arguments are used
- argument deduction might fail if ambiguous types are deduced
- the rules are highly complex... (see reference)

```
template <typename 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

- class template arguments can be deduced by the types of the arguments passed to a constructor

```cpp
#include <complex>

int main() {
    std::complex c1{1.1, 2.2}; // deduces to std::complex<double>
    std::complex c2{3.3};      // deduces to std::complex<double>
    std::complex c3{4, 5.5};   // ERROR: ambiguous deduction
}
```

- if class template argument deduction could be interpreted as initializing a copy, this is preferred:

```cpp
#include <vector>

int main() {
    std::vector v1{42}; // vector<int> with one element
    std::vector v2{v1}; // v2 is also vector<int>
                        // and not vector<vector<int>>
}
```

Deduction guides

- you can define specific deduction guides to provide additional (or fix existing) class template argument deductions
- for details, see reference

- example:

```cpp
template <typename T1, typename T2>
struct MyPair {
    T1 first;
    T2 second;
    MyPair(const T1& x, const T2& y) : first{x}, second{y} {}
};

// deduction guide for the constructor:
template <typename T1, typename T2>
MyPair(T1, T2) -> MyPair<T1, T2>;

int main() {
    MyPair p1{"hi", "world"}; // pair of const char*
}
```

Deduction guides: standard library example

- the standard library uses multiple deduction guides, for example:

```
// let std::vector<> deduce element type from initializing iterators:
namespace std {
    template <typename Iterator>
    vector(Iterator, Iterator)
      -> vector<typename iterator_traits<Iterator>::value_type>;
}

// this allows, for example:
std::set<float> s;
// ... fill s ...
std::vector v(s.begin(), s.end()); // OK, std::vector<float>
```

The auto type revisited

- auto can be used to deduce the type of a variable from its initializer
  - deduction follows the same rules as function template argument deduction
- example:

```cpp
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> intToStringMap;

    std::unordered_map<int, std::string>::iterator it1 =
        intToStringMap.begin(); // noone wants to read this..

    auto it2 = intToStringMap.begin(); // much better!
}
```

The auto type and modifiers

- auto does not require any modifiers (such as const, *, &) to work
- however, it can make code hard to understand and error prone, so all known modifiers should always be added to auto

```
const int* foo(); // using raw pointers for illustrative purposes only

int main() {
    // BAD:
    auto f1 = foo();            // auto is const int*
    const auto f2 = foo();      // auto is int*
    auto* f3 = foo();           // auto is const int

    // GOOD:
    const auto* f4 = foo();     // auto is int
}
```

The auto type and references

- auto is not deduced to a reference type
- this might incur unwanted copies, so always add all known modifiers 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
}
```

Contents

Generic lambdas

- lambda functions can also be generic by using the type `auto` for at least one of its arguments:

```cpp
using namespace std::string_literals;

auto twice = [] (auto x) { return x + x; };

int i = twice(2); // i == 4
std::string s = twice("hi"s); // s == "hihi"
```

- in fact, the compiler will internally create a templated `operator()` for the lambda closure's type, similar to:

```cpp
struct _some_unique_name {
    template <typename T>
    auto operator() (T x) { return x + x; }
};
```

- several `auto` arguments will translate into several template parameters, and you can mix and match with regular types

Templated lambdas

- since C++20, lambdas can also be templated:

```cpp
// generic lambda:
auto sumGen = [](auto x, auto y) { return x + y; };

// templated lambda:
auto sumTem = []<typename T>(T x, T y) { return x + y; };
```

- this allows more specific lambdas, such as:

```cpp
// generic lambda for generic container:
auto sizeGen = [](const auto& container)
    { return container.size(); };

// templated lambda for any std::vector:
auto sizeVec = []<typename T>(const std::vector<T>& vec)
    { return vec.size(); };
```

Functions with `auto` parameters

- in fact, since C++20, also ordinary functions can take `auto` parameters as well:

```
void f1(auto x) { /* ... */ }
```

- this automatically translates into a function template with an invented template parameter for each `auto` parameter
- this does not necessarily improve readability of code...

- this is an unconstrained `auto`, there is also a constrained version for C++20 concepts (see later)

Contents

Parameter packs

- parameter packs are template parameters that accept zero or more arguments
  - non-type: `type ... Args`
  - type: `typename|class ... Args`
  - template: `template < param_list > typename|class ... Args`
- parameter packs can appear in alias, class, and function templates
- templates with at least one parameter pack are called variadic templates
- function parameter packs appear in the parameter list of a variadic function template
  - syntax: `Args ... args`
- to expand a parameter pack to a comma-separated list:
  - syntax: `pattern ...`

Variadic templates: declaration example

```cpp
// a variadic class template with one parameter pack
template <typename... T>
struct Tuple { };

// a variadic function template, expanding the pack in its argument
template <typename... T>
void printTuple(const Tuple<T...>& tuple);

// a variadic function template with a function parameter pack
template <typename... T>
void printElements(const T&... args);

int main() {
    Tuple<int, int, float> tuple;

    printTuple(tuple);
    printElements(1, 2, 3, 4);
}
```

Variadic templates: implementation example

- implementation of variadic templates can be quite complex
    - one relatively straightforward way: tail recursion

```cpp
#include <iostream>

void printElements() { std::cout << "\n"; }

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

- fold expressions can be used to reduce a parameter pack over a binary operator op
  - variant 1 (unary left): `( ... op pack )`
  - variant 2 (unary right): `( pack op ... )`
  - variant 3 (binary left): `( init op ... op pack )`
  - variant 4 (binary right): `( pack op ... op init )`
  - `pack` must be an expression containing an unexpanded parameter pack
  - `init` must be an expression not containing a parameter pack
- semantics:
  - $(\ldots \circ E)$ becomes $((E_1 \circ E_2) \circ \ldots) \circ E_n$
  - $(E \circ \ldots)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ E_n))$
  - $(I \circ \ldots \circ E)$ becomes $(((I \circ E_1) \circ E_2) \circ \ldots) \circ E_n$
  - $(E \circ \ldots \circ I)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ (E_n \circ I)))$

Unary fold expression: example

- fold expressions enable more concise implementations of variadic templates in some cases:

```cpp
template <typename R, typename... Args>
R reduceSum(const Args&... args) {
    return (args + ...);
}

int main() {
    return reduceSum<int>(1, 2, 3, 4); // returns 10
}
```

- even though concise, the code can become very quickly very hard to understand

Binary fold expression: example

- a simple example for a binary fold expression:

```
template <typename... T>
void print(const T&... args) {
    (std::cout << ... << args) << '\n';
}

int main() {
    print();          // prints nothing
    print(1);         // prints "1\n"
    print(1, 2, 3);   // prints "123\n"
}
```

Variadic templates: implementation example revisited

- the previous implementation example can also be done using fold expressions
    - judge for yourself if it is any simpler/better...

```cpp
#include <iostream>

template <typename Head, typename... Tail>
void printElements(const Head& head, const Tail&... tail) {
    std::cout << head;

    // generic lambda
    auto outWithComma = [](const auto& arg) {
                                    std::cout << ", " << arg; };

    (..., outWithComma(tail) ); // unary fold expression

    std::cout << "\n";
}

int main() {
    printElements(1, 2, 3.0, 3.14, 4);
}
```

Contents

Two meanings of constant

- there are two meanings of constant in C++:
  - `const`: do not modify this object in this scope (variables, member functions)
  - `constexpr`: something that can be evaluated at compile-time

- a constant expression is an expression that a compiler can evaluate at compile-time
  - it can only use values that are known at compile-time (for example literals)
  - it can combine those values with operators and `constexpr` functions
  - it cannot have side effects (such as exceptions or memory allocations)

`constexpr` keyword

- the keyword `constexpr` declares a variable or function to be able to be evaluated at compile-time
- `constexpr` variables must be
  - of literal type (e.g. scalar, reference, or user-defined type with `constexpr` constructor/destructor)
  - immediately initialized with a constant expression
- `constexpr` functions must
  - have literal return types and arguments
  - not contain things like `goto` or variable definitions of non-literal types or static/thread storage duration
- for full details, see reference

const vs. constexpr examples

- a const variable
  - that is initialized with a constant expression can be used in a constant expression (without having to use constexpr)
  - but it can also be initialized by something that is not a constant expression!

```
int f(int x) { return x * x; }

constexpr int g(int x) { return x * x; }

int main() {
    const int x{7};       // constant expression
    const int y = f(x);   // not a constant expression!
    const int z = g(x);   // constant expression

    constexpr int xx{x}; // ok
    constexpr int yy{y}; // ERROR: f(x) not constant expression
    constexpr int zz{z}; // ok
}
```

constexpr std::vector and constexpr std::string

- since C++20, the standard library types `std::vector` and `std::string` can be used as `constexpr`

```
int main() {
    const std::string s{"Hello"};  // constexpr since C++20

    constexpr std::string ss{s};   // ok since C++20
}
```

- caveat: some compilers don't support this yet

constexpr functions

- constexpr functions are like regular functions, except they cannot have side effects
  - hence no goto and no usage of non-literal types or static/thread storage duration
  - since C++20, exceptions and exception handling may be used
- to make your user-defined type (class) a literal type, it needs
  - a constexpr destructor (or trivial before C++20)
  - to have at least one constexpr constructor (that is not copy/move), or it could be an aggregate type (see reference)
- member functions can be constexpr
  - if you want the member function to be const, you have to additionally declare it as such

```
struct A {
    constexpr void myFunc() const;
};
```

consteval functions

- constexpr functions can still be called at run-time (and will then execute at run-time), even though you can use them in constant expressions
- if you want guaranteed execution at compile-time, C++20 introduced the consteval keyword to enable immediate functions
  - consteval functions have the same restrictions as constexpr functions
  - you cannot mix consteval and constexpr
- example:

```cpp
consteval int sqr(int n) {
    return n * n;
}
```

consteval examples

```cpp
int sqrRunTime(int n) { return n * n; }

consteval sqrCompileTime(int n) { return n * n; }

constexpr sqrRunOrCompileTime(int n) { return n * n; }

int main() {
    constexpr int p1 = sqrRunTime(100); // ERROR: not constexpr
    constexpr int p2 = sqrCompileTime(100);
    constexpr int p3 = sqrRunOrCompileTime(100);

    int x{100};
    int p4 = sqrRunTime(x);
    int p5 = sqrCompileTime(x); // ERROR: x not constant expression
    int p6 = sqrRunOrCompileTime(x);

    int p7 = sqrCompileTime(100);       // compile-time
    int p8 = sqrRunOrCompileTime(100); // run-time or compile-time
}
```

Compile-time if

- with `if constexpr` we can evaluate conditions at compile-time

```
if constexpr ( init_statement; condition )
    statement_true
else
    statement_false
```

- `init_statement` is optional
- if the `condition` is true, then `statement_false` is discarded (no code generated, but it is still checked like code of unused templates), if `condition` is false, then `statement_true` is discarded (but not ignored, see later)
- the `else` branch is optional

Compile-time if: example

```cpp
#include <iostream>
#include <memory>      // for std::make_unique
#include <type_traits> // for std::is_pointer_v

template <typename T>
auto getValue(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t;  // deduces return type to int for T = int
}

int main() {
    int i = 9;
    auto j = std::make_unique<int>(9);

    std::cout << getValue(i) << "\n";       // output "9"
    std::cout << getValue(j.get()) << "\n"; // output "9"
}
```

Compile-time if: details

- the discarded statement is not ignored, it is checked for valid syntax and validity of all names (that do not depend on template parameters)
  - some compilers (e.g. MSVC) do not do this correctly, so they might compile code that is not standards-conform!
- there is an important difference to run-time if: compile-time if does not short-circuit the condition evaluation!
  - i.e. it does not stop evaluation of `&&` until the first `false` or of `||` until the first `true`
- you can use compile-time if outside templated code
  - the only real purpose of that is to reduce code size (because of the discarded statements)
  - however, opposed to compile-time if in templated code, the discarded statements still need to fully compile

Compile-time if: fancy example

- example by N. Josuttis for perfect return of a generic value:

```cpp
#include <functional>  // for std::forward
#include <type_traits> // for std::is_same and std::invoke_result

template <typename Callable, typename... Args>
auto call(Callable op, Args&&... args) {
    if constexpr (std::is_void_v<std::invoke_result_t<Callable, Args...>>) {
        // return type is void:
        op(std::forward<Args>(args)...);
        // do something before we return
        return;
    }
    else { // return type is not void:
        auto retValue{op(std::forward<Args>(args)...)};
        // do something with retValue before we return
        return retValue;
    }
}
```

constexpr lambdas

- by default, lambdas are implicitly constexpr if possible

```
auto squared = [] (auto x) { return x * x; }    ;

std::array<int, squared(5)> a; // ok, std::array<int, 25>,
                               // as squared(5) is constexpr
```

- to ensure that a lambda is constexpr, you can declare it as constexpr after the argument list, before the optional trailing return type:

```
auto squared2 = [] (auto x) constexpr { return x * x; };
auto squared3 = [] (int x) constexpr -> int { return x * x; };
```

Contents

Template meta programming

- templates are instantiated at compile-time
- this allows compile-time programming (or: template meta programming)
- templates are actually a Turing-complete (sub-)language
- there are extremely useful, but at times very complicated, tricks

```cpp
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<5>::value; // computes 5! at compile-time
}
```

Template meta programming: another example

- template specialization enables termination of recursion:

```cpp
#include <iostream>

template <unsigned N>
constexpr unsigned fibonacci() {
    return fibonacci<N-1>() + fibonacci<N-2>();
}

template <>
constexpr unsigned fibonacci<1>() { return 1; }

template <>
constexpr unsigned fibonacci<0>() { return 0; }

int main() {
    std::cout << fibonacci<10>() << "\n"; // computes compile-time
}
```

Template meta programming with `if constexpr`

- `if constexpr` can simplify template meta programming a lot, making it read similar to regular code:

```cpp
#include <iostream>

template <unsigned N>
constexpr unsigned fibonacci() {
    if constexpr (N >= 2)
        return fibonacci<N-1>() + fibonacci<N-2>();
    else
        return N;
}

int main() {
    std::cout << fibonacci<10>() << "\n"; // computes compile-time
}
```

static_assert

- static_assert 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 is a string that appears as a compiler error if bool_constexpr is false

```
template <unsigned N>
class NonEmptyBuffer {
    static_assert(N > 0, "buffer size must be positive");
};

int main() {
    NonEmptyBuffer<0> buffer; // ERROR: static assertion failed
}
```

Type traits

- type traits compute information about types at compile-time
    - for example: `std::numeric_limits` is a type trait
    - there are many useful traits in the standard library (see reference)
- small type trait example:

```
template <typename T>
struct  IsUnsigned {
    static constexpr bool value = false;
};

template <>
struct IsUnsigned <unsigned char> {
    static constexpr bool value = true;
};
/* add further specializations for all other unsigned types... */

template <typename T>
void foo() {
    // make sure template argument is unsigned
    static_assert(IsUnsigned<T>::value);
}
```

SFINAE

- SFINAE stands for “Substitution Failure Is Not An Error”
  - when compiling templated overload candidates, the specified (or deduced) types are substituted into the template arguments
  - this can lead to non-sensical code (failure), which in this case will not produce an error
- simple example (by E. Bendersky):

```
int negate(int i) { return -i; }

template <typename T>
typename T::value_type negate(const T& t) { return -T(t); }

int main() {
    negate(42); // int negate(int) will be called
                // templated function is a substitution failure
}
```

SFINAE and and `std::enable_if`

- SFINAE and a helper struct `std::enable_if` allow you to restrict template parameters:

```cpp
template <typename T,
          typename std::enable_if<std::is_arithmetic_v<T>,
                                  bool>::type = true>
T f(T t) { return 2*t; }
```

- alternatively as type template parameter:

```cpp
template <typename T,
          typename = std::enable_if_t<std::is_arithmetic_v<T>>>
T f(T t) { return 2*t; }
```

  - careful though, this version can be circumvented by invoking it with an explicit second argument: `f<A, void>(a);`

- alternatively via return type:

```cpp
template <typename T>
typename std::enable_if_t<std::is_arithmetic_v<T>, T>
f(T t) { return 2*t; }
```

How std::enable_if works

- a possible implementation of std::enable_if is

```cpp
template<bool B, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> { using type = T; };

// for shorter notation:
template<bool B, typename T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

- usage from previous example:

```cpp
template <typename T,
          typename std::enable_if<std::is_arithmetic_v<T>,
                                  bool>::type = true>
T f(T t) { return 2*t; }
```

Bonus: variadic SFINAE with fold expressions

- you can combine SFINAE and fold expressions (credit to N. Josuttis):

```cpp
// check if passed types are homogeneous
template <typename T1, typename... TN>
struct IsHomogeneous {
    static constexpr bool value = (std::is_same_v<T1,TN> && ...);
};

// check if passed parameters have same type
template <typename T1, typename... TN>
constexpr bool isHomogeneous(T1, TN...) {
    return (std::is_same_v<T1,TN> && ...);
}

int main() {
    std::cout << IsHomogeneous<int, std::size_t>::value << "\n"; // outputs "0\n"
    std::cout << isHomogeneous(43, -1) << "\n";                  // outputs "1\n"
}
```

C++20 concepts

- a better way to express restrictions on template parameters was introduced in C++20: concepts

```
#include <concepts>

template <typename T>
requires std::integral<T>
T gcd1(T a, T b)
{ if (b==0) return a; else return gcd1(b, a%b); }

template <typename T>
T gcd2(T a, T b) requires std::integral<T>
{ if (b==0) return a; else return gcd2(b, a%b); }

template <std::integral T>
T gcd3(T a, T b)
{ if (b==0) return a; else return gcd3(b, a%b); }

std::integral auto gcd4(std::integral auto a, std::integral auto b)
{ if (b==0) return a; else return gcd4(b, a%b); }
```

- more on this later...

Contents

Numerical linear algebra

- many applications use numerical linear algebra, i.e. operations with matrices and vectors
- typical operation example:

$$z = c_1 * x + c_2 * y$$

  where x, y, z are vectors of size n, and c_1, c_2 are scalar

- standard solution: use an existing library
  - for example: Eigen, Boost uBLAS
  - there are standardization efforts (Linear Algebra TS), but they are not expected before C++23 (or later...)

Operation implementation example

$$z = c_1 * x + c_2 * y$$

- C-style implementation using indexed arrays:

```
for (int i = 0; i < n; ++i)
    z[i] = c1 * x[i] + c2 * y[i];
```

- C++-style implementation using classes and overloaded operators:

```
class vector { /* ... */ };
vector operator+(const vector& lhs, const vector& rhs);
vector operator*(double lhs, const vector& rhs);

z = c1 * x + c2 * y;
```

Operation implementation example: efficiency

- the C++-style implementation is many times slower than the C-style one
- why?
  - the compiler creates temporaries for the intermediate results, similar to this:

```
vector z1 = c1 * x;
vector z2 = c2 * y;
vector z3 = z1 + z2;
z = std::move(z3);
```

- also highly problematic: the additional memory required for those temporaries
- solution: expression templates!

Operator implementation: best of both worlds?

- can we have both the performance of the C-style implementation

```
for (int i = 0; i < n; ++i)
    z[i] = c1 * x[i] + c2 * y[i];
```

- and the elegance of the C++-style implementation?

```
z = c1 * x + c2 * y;
```

- and potentially gain even more performance?

- yes, using expression templates!
  - use compile-time programming to transform C++-style to C-style code
  - and add extras (e.g. SIMD vectorization, parallelization, GPU kernels)

Expression templates: the idea

- the general idea of expression templates is the following:
    - instead of evaluating each intermediate step into a temporary, build an expression tree
    - only when assigning the result, perform the actual computation by traversing the expression tree

- various implementations are possible for this, the most efficient ones use compile-time programming
    - already possible with C++98 (see Eigen library), but very complex and cumbersome
    - much more convenient with C++17 (see following)
    - the following slides are adapted from Bowie Owen's CppCon 2019 talk "Expression Templates for Efficient, Generic Finance Code"

Expressions and operators

- overloaded operators now return templated expressions:

```cpp
using add = /* ... */;
using mul = /* ... */;

template <typename op, typename... operands>
class Expr { /* ... */ };

Expr<add, vector, vector>
operator+(const vector& lhs, const vector& rhs);

Expr<mul, double, vector>
operator*(double lhs, const vector& rhs);
```

- these will be the leafs of the expression tree

Expression temporaries

- instead of temporary `vectors`, we now store temporary expressions (which are lightweight):

```
// no computations done here:
Expr<mul, double, vector> z1 = c1 * x;
Expr<mul, double, vector> z2 = c2 * y;
Expr<add,
    Expr<mul, double, vector>,
    Expr<mul, double, vector>> z3 = z1 + z2;

// all computations done in assignment:
vector z = z3;
```

Expression temporaries (cont.)

- calling the operators will create the expression tree for us implicitly:

```cpp
        // all temporaries in one step:
        Expr<add,
            Expr<mul, double, vector>,
            Expr<mul, double, vector>> z3 = c1 * x + c2 * y;
        // auto z3 = ...; would be more concise (but hide the expression tree)

        // all computations done in assignment:
        vector z = z3;
```

- the assignment operator of the `vector z` has to do all the work (also known as lazy evaluation)

Expression temporaries (cont.)

- regular code will then look like this:

```
        // everything in one step:
        vector z = c1 * x + c2 * y;
```

- the right-hand side of the assignment is the temporary expression tree
- the assignment operator of the `vector z` does all the work (lazy evaluation)

- side note: you now need to be careful of `auto z = ...`, as this will just store the expression tree and not evaluate the result

Implementing expression templates

- the expression needs to store the necessary information:
    - the operation (add, mul, etc.) as a callable object, to be called by the assignment operator later
    - references to the operands (vectors, scalars, other expressions)

```cpp
template <typename op, typename... operands>
class Expr {
    op callable_;
    std::tuple<const operands&...> args_;

public:
    Expr(op callable, const operands&... args)
        : callable_{callable}, args_{args...} {}
};
```

Implementing operators

- the overloaded operators now only record all the information as expressions:

```
template <typename LHS, typename RHS>
auto operator*(const LHS& lhs, const RHS& rhs) {
    return Expr{
        [] (auto const& l, auto const& r) {
                return l * r; },
        lhs, rhs
    };
}
```

- the template types for the expression of `c1 * x` would be `LHS==double` and `RHS==vector`
- the callable operation (the generic lambda) will be called element-wise on the operands
    - e.g. `l` and `r` would be `double`

Implementing assignment

- the actual computation is triggered in the assignment operator of the `vector` class
    - it will be performed element-wise
    - (this could be optimized to use vectorization, parallelization etc.)

```cpp
class vector {
    std::vector<double> v_;

public:
    // ...

    template <typename srctype>
    vector& operator=(const srctype& src) {
        for (unsigned i = 0; i < v_.size(); ++i)
            v_[i] = src[i];

        return *this;
    }
};
```

- now we need an `operator[]` to access the i-th element of our expression `src`

Extending the expression

- we extend the expression by `operator[]`:

```cpp
template <typename op, typename... operands>
class Expr {
    op callable_;
    std::tuple<const operands&...> args_;

public:
    // ...

    auto operator[](unsigned i) const {
        const auto call_at_index =
            [this, i](const operands&... a) {
                return callable_(subscript(a, i)...);
            };

        return std::apply(call_at_index, args_);
    }
};
```

- why not `callable_(a[i])`? (see next)

Checking if subscript is valid

- we need to check if our expression can handle a subscript
    - scalars can be operands of our expression
    - scalars cannot have subscripts and would cause a compile error

```
template <typename operand>
auto subscript(const operand& v, unsigned i) {
    if constexpr (is_vector_or_expression<operand>) {
        return v[i];
    }
    else {
        return v;
    }
}
```

- `is_vector_or_expression` is a type trait (see next)

Type trait: vector or expression

- the type trait for `subscript` can be defined like this:

```cpp
template <typename>
struct is_vector : std::false_type {};

template <>
struct is_vector<vector> : std::true_type {};

template <typename>
struct is_expression : std::false_type {};

template <typename op, typename... operands>
struct is_expression<Expr<op, operands...>> : std::true_type {};

template <typename T>
constexpr bool is_vector_or_expression
                    = is_vector<T>() || is_expression<T>();
```

Being careful about operands

- having arbitrary types as arguments to `operator*` can be problematic
- we can restrict them to only reasonable choices using type traits:

```
template <typename LHS, typename RHS>
constexpr bool is_binary_op_ok = /* ... */;
```

- restrict using SFINAE (substitution failure is not an error):

```
template <typename LHS, typename RHS,
    typename = std::enable_if_t<is_binary_op_ok<LHS, RHS>> >
auto operator*(const LHS& lhs, const RHS& rhs) { /* ... */ }
```

- more elegant using concepts from C++20:

```
template <typename LHS, typename RHS>
    requires(is_binary_op_ok<LHS, RHS>)
auto operator*(const LHS& lhs, const RHS& rhs) { /* ... */ }
```

Type trait: binary operator

- the type trait for `operator*` can be defined like this (using the previous type traits):

```cpp
template <typename T>
constexpr bool is_arithmetic = std::is_arithmetic_v<T>;

template <typename LHS, typename RHS>
constexpr bool is_binary_op_ok =
    (is_vector_or_expression<LHS> && is_vector_or_expression<RHS>)
    || (is_vector_or_expression<LHS> && is_arithmetic<RHS>)
    || (is_arithmetic<LHS> && is_vector_or_expression<RHS>);
```

Expression templates: summary

- using expression templates we can use C++-style numerical linear algebra, like:

```
            z = c1 * x + c2 * y;
```

- there is no overhead compared to the C-style implementation explicitly iterating over arrays
  - there is no extra memory required for temporaries
  - if you implemented vectorization etc. in `operator=`, it will even be faster
- now linear algebra can be conveniently and efficiently be used like mathematical formulas

Expression template implementations

- if you want to see expression templates done the C++98 way, check out the very popular Eigen library
  - `http://eigen.tuxfamily.org/`
- self-advertisement: you can check out our C++17 expression template implementation in our project “elsa”
  - CPU and GPU expression templates
  - `https://gitlab.lrz.de/IP/elsa`

Contents

Dynamic vs. static polymorphy

- dynamic polymorphy with virtual functions works very well, but it incurs a run-time cost
  - calls of virtual functions incur an indirection (vtable) and cannot be inlined
- static polymorphy tries to achieve the same without the additional runtime cost
  - this can be done using the Curiously Recurring Template Pattern (CRTP)

CRTP

- the idea is to inherit from a template class, with the derived class as the template parameter of the base class:

```
template <typename T>
class Base {
    // ...
};

class Derived : public Base<Derived> {
    // ...
};
```

- then we know the type of the derived class statically, and can use it to cast the base class into the derived class:

```
template <typename T>
class Base {
public:
    void doSomething() {
        T& derived = static_cast<T&>(*this);
        // use derived somehow
    }
};
```

CRTP: caveats

- while CRTP works, it is tricky to use and can lead to insidious errors
- for example: since there are no virtual functions, there is no overriding
  - instead, methods of the same name in the derived class will simply hide base class methods!

```
class Derived : public Base<Derived> {
public:
    void doSomething(); // this hides doSomething()
                        // from the base class!
};
```

- guideline: only use it where really necessary
  - for example in performance critical code (see Eigen library)
  - some special use cases such as polymorphic clone (see later)

CRTP: a simple example

- a simple example for static polymorphy:

```
template <typename T>
class Amount {
public:
    double getValue() const {
        return static_cast<const T&>(*this).getValue();
    }
};
```

- an implementation of the static interface:

```
class Constant42 : public Amount<Constant42> {
public:
    double getValue() const { return 42; }
};
```

CRTP: a simple example (cont.)

- another implementation of the static interface:

```cpp
class Variable : public Amount<Variable> {
public:
    explicit Variable(int value) : value_{value} {}

    double getValue() const { return value_; }

private:
    int value_;
};
```

CRTP: a simple example (cont.)

- using the static polymorphic interface:

```cpp
template <typename T>
void print(const Amount<T>& amount) {
    std::cout << amount.getValue() << ", ";
}

int main() {
    Constant42 c42;
    print(c42);
    Variable v(43);
    print(v);
}
// outputs 42, 43, as expected
```

- this works without any virtual function calls, at the cost of having templates in any code using static polymorphy
    - with C++20, concepts can help make it easier to use

Contents

The problem of polymorphic cloning

- when using polymorphic class hierarchies, copying objects using a base class reference/pointer is impossible:

```
class Abstract {
public:
    virtual void doStuff() = 0;
    virtual ~Abstract() = default;
};

class Concrete : public Abstract {
public:
    void doStuff() override { /* ... */ }
};

void f(Abstract& a) {
    Abstract& b = ??; // cannot copy a, as actual type unknown
}
```

Polymorphic cloning: classical solution

- idea: instead of constructors (which won't work), use a virtual clone function to delegate copying to the derived class:

```cpp
class Abstract {
public:
    virtual Abstract* clone() const = 0;

    virtual void doStuff() = 0;
    virtual ~Abstract() = default;
};

class Concrete : public Abstract {
public:
    Concrete* clone() const override {
        return new Concrete(*this);
    }

    void doStuff() override { /* ... */ }
};
```

  - using covariant return types!

Polymorphic cloning: classical solution (cont.)

- example usage:

```
void f(Abstract& a) {
    Abstract* b = a.clone(); // now it works!
}
```

- however, there is a problem: memory management and ownership!
- solution: use smart pointers and RAII, i.e. `std::unique_ptr`

Polymorphic cloning with smart pointers

- simply return `std::unique_ptr` instead of raw pointers:

```cpp
class Abstract {
public:
    virtual std::unique_ptr<Abstract> clone() const = 0;

    virtual void doStuff() = 0;
    virtual ~Abstract() = default;
};

class Concrete : public Abstract {
public:
    std::unique_ptr<Abstract> clone() const override {
        return std::make_unique<Concrete>(*this);
    }

    void doStuff() override { /* ... */ }
};
```

- unfortunately, a covariant return type is no longer possible, as it only works with raw pointers/references

Polymorphic cloning with smart pointers and covariance

- we can enable polymorphic cloning with smart pointers and covariance with a trick:

```cpp
class Abstract {
public:
    std::unique_ptr<Abstract> clone() const {
        return std::unique_ptr<Abstract>(this->cloneImpl());
    }
    virtual ~Abstract() = default;
private:
    virtual Abstract* cloneImpl() const = 0;
};

class Concrete : public Abstract {
public:
    std::unique_ptr<Concrete> clone() const {
        return std::unique_ptr<Concrete>(this->cloneImpl());
    }
private:
    Concrete* cloneImpl() const override {
        return new Concrete(*this);
    }
};
```

Polymorphic cloning

- “abusing” name hiding, `clone()` hides the base class method, enabling to use the correct smart pointer type
- the actual work is done in the private virtual `cloneImpl()` method that uses a covariant return type

- drawback: in every class you have to define `clone()` and `cloneImpl()`
- solution: use CRTP to inject the two functions

Polymorphic cloning: the Cloneable pattern

- the abstract base class `Cloneable` providing the interface:

```cpp
class Cloneable {
public:
    virtual ~Cloneable() {} // required!

    std::unique_ptr<Cloneable> clone() const {
        return std::unique_ptr<Cloneable>(this->cloneImpl());
    }

private:
    virtual Cloneable* cloneImpl() const = 0;
};
```

- side-note: if you want to implement this pattern using `std::shared_ptr`, you need to take extra care (see `std::enable_shared_from_this`)

Polymorphic cloning: the Cloneable pattern (cont.)

- injecting the boilerplate code using CRTP:

```cpp
template <typename Derived, typename Base>
class CloneInherit : public Base {
public:
    std::unique_ptr<Derived> clone() const {
        return std::unique_ptr<Derived>(
            static_cast<Derived*>(this->cloneImpl()));
    }

private:
    virtual CloneInherit* cloneImpl() const override {
        return new Derived(static_cast<const Derived&>(*this));
    }
};
```

Polymorphic cloning: the Cloneable pattern (cont.)

- now concrete derived classes need to do nothing besides inheriting the CRTP class:

```
class Concrete : public CloneInherit<Concrete, Cloneable> {
    // nothing to be done!
};
```

- using it:

```
int main() {
    auto c = std::make_unique<Concrete>();
    auto cc = c->clone();

    std::cout << "cc is a " << typeid(cc).name() << "\n";
}
```

The C++ standard library

- a collection of useful C++ classes, types, and functions
  - everything is declared within the `std` namespace
  - implemented in C++

- part of the ISO C++ standard
  - defines interface, semantics, and contracts the implementation has to deliver (e.g. complexity guarantees)
  - implementation is *not* part of the standard!
  - multiple implementations exist, e.g. `libstdc++` (used by gcc) and `libc++` (used by llvm/clang)

- functionality is divided into sub-libraries, with multiple headers each
- parts of the C standard library are included for backwards compatibility
  - if you really cannot avoid it

The C++ standard library: feature overview

Overview over the most important library features:

- utilities
  - memory management (`unique_ptr`, `shared_ptr`)
  - error handling (exceptions, `assert`)
  - time (clocks, durations, timestamps)
  - optionals, variants, pairs, tuples, ...
  - type traits
- strings (`string` class, string views, C-style string handling)
- containers: `array`, `vector`, lists, maps, sets, ...
- algorithms: sort, search, max, min, ...
- iterators
- numerics
  - common mathematic functions (`sqrt`, `pow`, `mod`, `log`, ...)
  - complex numbers
  - random number generation

The C++ standard library: feature overview (cont.)

- input/output
    - input/output streams
    - string streams, file streams
    - formatting library
    - filesystem library
- threading
    - thread classes
    - coroutines
    - mutexes
    - semaphores
    - latches and barriers
    - atomics
- and many more
    - regex
    - localization
    - calendar and time zones
    - ...

Contents

`std::optional`

- `std::optional` encapsulates a value that might or might not exist
- defined in `<optional>` header
- template parameter `T` denotes the type of the value the `std::optional` might contain
    - `std::optional<int>` might contain an `int`
- ideal as a return type for functions that might not produce a valid result
    - if the computation succeeded, return an `std::optional` containing a value
    - if it failed, return an `std::optional` without a value
- it is an object, despite support dereferencing `*` and `->`
    - internally implemented as an object with a member of type `T` and a boolean

std::optional example

- creation through constructor or std::make_optional

```cpp
std::optional<std::string> mightFail(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)
        return std::make_optional<std::string>("less than 7");
    else
        return std::nullopt; // alternatively: return {};
}
```

- value of an optional can be read with value() (throws exception when empty) or dereferenced with * or -> (undefined behavior when empty)

```cpp
mightFail(3).value(); // "less than 7"
mightFail(8).value(); // throws std::bad_optional_access

*mightFail(3); // "less than 7"
mightFail(6)->size(); // 11
mightFail(7)->empty(); // undefined behavior
```

std::optional example (cont.)

- checking if an optional has a value

```
mightFail(3).has_value(); // true
mightFail(8).has_value(); // false

auto opt5 = mightFail(5);
if (opt5) // contextual conversion to bool
    opt5->size(); // 11
```

- providing a default value

```
mightFail(42).value_or("default"); // "default"
```

- clearing an optional:

```
auto opt6 = mightFail(6);
opt6.has_value(); // true
opt6.reset(); // clears the value
opt6.has_value(); // false
```

Pair

- `std::pair<T, U>` is a template class that stores exactly one object of type `T` and one of type `U`
  - defined in header `<utility>`
  - constructor takes object of `T` and `U`
  - pairs can also be constructed with `std::make_pair()`
  - objects can be access with `first` and `second`
  - can be compared for equality/inequality
  - can be compared lexicographically with <, <=, >, >=

```
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
```

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 header `<tuple>`
    - constructor takes all objects
    - tuples can also be constructed with `std::make_tuple()`
    - the ith object can be accessed with `std::get<i>()`
    - just like pairs, tuples define all relational comparison operations
- structured bindings work perfectly for decomposing tuples (and pairs)

```cpp
auto t = std::make_tuple(123, 4.56);
auto [a, b] = t; // a is type int, b is type double

auto p = std::make_pair(4, 5);
auto& [x, y] = p; // x, y have type int&
x = 123; // p.first is now 123
```

Contents

Strings

- `std::string` encapsulates character sequences
    - manages its own memory (RAII)
    - guaranteed contiguous memory storage
    - knows its length
    - wide array of member functions/operators for string manipulation
    - can be used like a C-style char pointer
    - since C++20, it has `constexpr` constructors
- defined in `<string>` header
    - alias for `std::basic_string<char>`
    - specializations for other character types exist (UTF)
    - you can do your own specializations for custom character types
- always prefer `std::string` over char pointers!

Using `std::string`

- default constructor creates empty string of length 0

```
std::string s;
s.size(); // == 0
```

- construct from string literal

```
std::string s_c1{"a string"};
std::string s_c2 = "another string";
```

- access single characters like an array

```
std::string s{"Hello World!");
std::cout << s.at(4) << s[6] << "\n"; // prints "oW"
```

- `at()` is bounds checked, `[]` is not, both return a reference

```
s.at(4) = 'x';
s[6] = 'Y';
std::cout << s << "\n"; // prints "Hellx Yorld!"
```

Using std::string (cont.)

- iterate over string contents:

```cpp
std::string s{"Hello World!"};
for (auto& c : s) // range-for
    ++c;
std::cout << s << "\n"; // prints Ifmmp!Xpsme"

// explicit iterator use
for (auto iter = s.begin(); iter != s.end(); ++iter)
    ++(*iter);
std::cout << s << "\n"; // prints Jgnnq"Yqtnf#
```

- comparing strings lexicographically

```cpp
std::string u0510{"breezy badger"};
std::string u1804{"bionic beaver"};
std::string u2004{"focal fossa"};

assert(u2004 > u1804); // okay, f after b
assert(u1804 > u0510); // fails, bi before br. Why, Ubuntu?!
```

`std::string` operations

- there are many more `std::string` operations, such as
  - `size()` or `length()`: the number of characters in the string
  - `empty()`: returns true if string has no characters
  - `append()` and +=: appends another string or character
  - +: concatenates strings
  - `find()`: returns 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. For read-only access, `std::string_view` is preferred (see next slide)

`std::string_view`

- creating substrings of an `std::string` via `substr()` is expensive (data is copied)
  - can be huge overhead (e.g. parsing large JSON files)
- `std::string_view` provides a read-only view on already existing strings
  - creation, substring and copying in constant time
  - internally: just a pointer and a length
  - defined in `<string_view>` header
- creation: from `std::string`, or from a char pointer with a length, or from a string literal
  - beware lifetime issues!
- has all read-only member functions of `std::string`
- `substr()` creates another string view in $O(1)$

Using std::string_view

```cpp
std::string s{"garbage garbage garbage interesting garbage"};

std::string sub = s.substr(24, 11); // with string: O(n)

std::string_view s_view{s}; // with string_view: O(1)
std::string_view sub_view = s_view.substr(24, 11); // O(1)

// in place operations:
s_view.remove_prefix(24); // O(1)
s_view.remove_suffix(s_view.size() - 11); // O(1)
```

```cpp
// useful for function calls
bool isEqNaive(std::string a, std::string b) { return a == b; }
bool isEqWithViews(std::string_view a, std::string_view b)
    { return a == b; }

isEqNaive("abc", "def"); // 2 allocations at runtime
isEqWithViews("abc", "def"); // no allocation at runtime
```

Contents

Containers: an 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 the elements stored are template parameter(s)
- provide member functions for direct element access, or through iterators
- (most) member functions shared between containers
- guarantees about the complexity of their operations:
    - sequence containers: optimized for sequential access (e.g. `std::array`, `std::vector`, `std::list`)
    - associative containers: sorted, optimized for search ($O(\log n)$) (e.g. `std::set`, `std::map`)
    - unordered associative containers: hashed, optimized for search (amortized $O(1)$, worst case $O(n)$) (e.g. `std::unordered_set`, `std::unordered_map`)

`std::vector`

- vectors are arrays that can dynamically grow in size
- defined in header `<vector>`
- elements are stored contiguously, and can be inserted/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()`
- since C++20: has `constexpr` constructors
- time complexity:
  - random access: $O(1)$
  - insertion and removal at end: typically $O(1)$, worst case: $O(n)$ due to possible reallocation
  - insertion and removal at any other position: $O(n)$

std::vector example

```cpp
std::vector<int> fib{1, 1, 2, 3};
fib.at(0); // == 1
fib[3] = 43;
fib.at(2) = 42; // fib is now 1, 1, 42, 43

// inserting or removing elements at end
fib.push_back(5); // fib is now 1, 1, 42, 43, 5
int myFib = fib.back(); // == 5
fib.pop_back(); // fib is now 1, 1, 42, 43

// inserting or removing elements anywhere with an iterator
auto it = fib.begin(); it += 2;
fib.insert(it, 41); // fib is now 1, 1, 41, 42, 43

// insertion invalidated the iterator, get a new one
it = fib.begin(); it += 2;
fib.erase(it); // fib is now again 1, 1, 42, 43

// clear the vector
fib.clear();
fib.empty(); // == true
fib.size(); // == 0
```

std::vector example (cont.)

```cpp
struct ExpensiveToCopy {
    ExpensiveToCopy(int id, std::string comment)
        : id_{id}, comment_{comment} {}
    int id_;
    std::string comment_;
};

std::vector<ExpensiveToCopy> vec;

// the expensive way:
ExpensiveToCopy e1(1, "my comment 1");
vec.push_back(e1); // needs to copy e1!
// better: use move
vec.push_back(std::move(e1));

// the best way: construct element in place
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 example (cont.)

- if final size of vector is known, give the vector a hint to avoid unnecessary reallocations:

```
std::vector<int> vec;
vec.reserve(1000000); // allocate space for 1000000 elements
vec.capacity(); // == 1000000
vec.size(); // == 0 (no elements stored yet!)

for (int i = 0; i < 1000000; ++i)
    vec.push_back(i); // no reallocations in this loop
```

- if the vector won't grow in the future, you can release unused space

```
std::vector<int> vec;
vec.reserve(100);

for (int i = 0; i < 10; ++i)
    vec.push_back(i);

// only needed a capacity of 10, free the space for the rest
vec.shrink_to_fit();
```

Other containers

- `std::array` for compile-time fixed-size arrays
- `std::deque` is a double-ended queue
  - basically a `std::vector` that is not guaranteed to be contiguous in memory
- `std::list` is a doubly-linked list
- `std::forward_list` is a singly-linked list

Container adaptors

- `std::queue` provides the typical queue interface, wrapping one of the sequence containers (array, vector, deque, list, forward_list)
- `std::stack` provides the typical stack interface, wrapping one of the sequence containers
- `std::priority_queue` provides a typical priority interface, wrapping one of the sequence containers with random access (vector, deque)

`std::unordered_map`

- unordered maps are associative containers of key-value pairs
- defined in header `<unordered_map>`
- keys are required to be unique
- at least two template parameters: `Key` and `T` (type of 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)
- if desired, accepts custom hash- and comparison functions through third and fourth template parameter

std::unordered_map example

- construction using pairs

```cpp
std::unordered_map<std::string, double>
   nameToGrade {{"maier", 1.3}, {"huber", 2.7}, {"lasser", 5.0}};
```

- value lookup using at() or []

```cpp
nameToGrade["huber"]; // == 2.7
nameToGrade["lasser"] = 4.3;
```

- search for a key

```cpp
auto search = nameToGrade.find("maier");
if (search != nameToGrade.end()) {
    // returns an iterator pointing to a pair
    search->first; // == "maier"
    search->second; // == 1.3
}
```

- to check if a key exists

```cpp
nameToGrade.count("blafasel"); // == 0
nameToGrade.contains("blafasel"); // == false (since C++20)
```

std::unordered_map example (cont.)

- updating or inserting elements

```cpp
nameToGrade["newStudent"] = 1.0;
```

- you can also insert/emplace

```cpp
auto p = std::make_pair("mueller", 2.0);
nameToGrade.insert(p);

// or simpler:
nameToGrade.insert({"mustermann", 4.0});

// or emplace:
nameToGrade.emplace("gruber", 1.7);
```

- remove elements

```cpp
auto search = nameToGrade.find("lasser");
nameToGrade.erase(search); // remove pair with "lasser" as key

nameToGrade.clear(); // removes all elements
```

std::map

- in `std::map` the keys are sorted
- defined in header `<map>`
- interface largely the same as for `std::unordered_map`
- is internally a tree (usually AVL or red-black)
- $O(\log n)$ complexity for random access, search, insertion, and removal
- `std::map` also allows to search for ranges

```cpp
std::map<int, int> xToY {{1, 1}, {3, 9}, {7, 49}};
// returns an iterator to first greater element:
auto gt3 = xToY.upper_bound(3);

while (gt3 != xToY.end()) {
    std::cout << gt3->first << "->" << gt3->second << "\n"; // 7->49
    ++gt3;
}

// returns iterator to first element not lower
auto geq3 = xToY.lower_bound(3); // 3->9 , 7->49
```

`std::unordered_set`

- sets are associative containers consisting of keys
- defined in header `<unordered_set>`
- keys are required to be unique
- template parameter `Key` for the type of 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 become corrupted
- if desired, accepts hash- and comparison-functions through second and third template argument

std::unordered_set example

- construction like arrays:

```cpp
std::unordered_set<std::string>
    shoppingList {"milk", "bread", "butter"};
```

- find an element

```cpp
auto search = shoppingList.find("milk");

if (search != shoppingList.end()) {
    // returns an iterator to the element
    *search; // == "milk"
}
```

- or determine if element is there

```cpp
shoppingList.count("bread"); // == 1
shoppingList.contains("blafasel"); // == false (since C++20)
```

std::unordered_set example (cont.)

- inserting elements

```
            shoppingList.insert("lettuce");
            shoppingList.emplace("lightbulb");
```

- insert returns a std::pair<iterator, bool> indicating if insertion succeeded

```
            auto result = shoppingList.insert("milk");

            result.second; // == false, as "milk" already in set
            *result.first; // == "milk", iterator points to element

            result = shoppingList.insert("broccoli");
            result.second; // == true, "broccoli" was added
            *result.first; // == "broccoli", iterator points to new element
```

- removing elements

```
            auto search = shoppingList.find("milk");
            shoppingList.erase(search); // "milk" is gone

            shoppingList.clear(); // now the set is empty
```

std::set

- in `std::set` the elements are sorted (in contrast to unordered sets)
- defined in header `<set>`
- interface largely the same as `std::unordered_set`
- is internally a tree (usually AVL or red-black)
  - $O(\log n)$ complexity for random access, search, insertion, and removal
- `std::set` also allows to search for ranges

```cpp
std::set<int> x {1, 3, 7};
auto gt3 = x.upper_bound(3);

while (gt3 != x.end()) {
    std::cout << x << "\n"; // 7
    ++gt3;
}

auto geq3 = x.lower_bound(3); // 3, 7
```

Containers: thread safety

- containers have some thread safety guarantees:
  - two different containers do not share state, i.e. all member functions can be called concurrently by different threads
  - the same container: all const member functions can be called concurrently. `at()`, `[]` (except in associative containers), `front()`, `back()`, `begin()`, `end()`, `find()` also count as const
  - 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
- rule of thumb: simultaneous reads on the same container are always OK, simultaneous read/writes on different containers are also OK. Everything else requires synchronization!

Contents

Iterators: overview

- Problem: different element access method for each container, i.e. container types not easily exchangeable in code
- Solution: iterators abstract over element access and provide pointer-like interface, allowing for easy exchange of underlying container type

- the standard library defines multiple iterator types for containers with different capabilities (random access, traversable in both directions, etc.)

- careful about iterator invalidation!
    - when writing to a container, all existing iterators are invalidated and can no longer be used! (except in exceptional cases...)

Iterator example

- all containers have a `begin()` and `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 to point at the next element

```
++it; // always prefer pre-increment
std::cout << *it; // prints "two"
```

Iterator example (cont.)

- iterators can be checked for equality; comparing to the `end()` iterator is used to check if iteration is done

```
while (it != end) {
    std::cout << *it << ",";
    ++it;
} // prints "three,four,"
```

- range-for loop can also be used to iterate over a container; it requires the range expression to have a `begin()` and `end()` iterator

```
for (auto elem : vec)
    std::cout << elem << ",";
// prints "one,two,three,four,"
```

Iterator example (cont.)

- iterators also help with dynamic insertion/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(); ) {
    if (it->size() == 3) {
        it = vec.erase(it);
        // erase returns a new, valid iterator
        // pointing to the next element
    }
    else
        ++it;
}
// vec == {"three", "four"}
```

InputIterator and OutputIterator

- InputIterator and OutputIterator are the most basic iterators
  - equality comparison: checks if two iterators point to the same position
  - dereferencable with * and ->
  - incrementable to point at the next element in sequence
  - a dereferenced InputIterator can only be read
  - a dereferenced OutputIterator can only be written to
- some limitations as the most basic iterators:
  - single-pass only: they cannot be decremented
  - only allow equality comparison (no <, >= etc.)
  - can only be incremented by one (i.e. it+2 does not work)
- typical use cases: single-pass algorithms like find() with InputIterator, or copy(), copying from IntputIterator to OutputIterator

`ForwardIterator` and `BidirectionalIterator`

- `ForwardIterator` combines `InputIterator` and `OutputIterator`
    - all the features and restrictions shared between input- and output iterator apply
    - dereferenced iterator can be read and written to

- `BidirectionalIterator` generalizes `ForwardIterator`
    - additionally allows decrementing (walking backwards)
    - supports multi-pass algorithms traversing the container multiple times
    - all other restrictions of `ForwardIterator` still apply

RandomAccessIterator and ContiguousIterator

- RandomAccessIterator generalizes BidirectionalIterator
  - additionally supports random access with operator[]
  - supports relational operators, such as < or >=
  - can be incremented or decremented by any amount (i.e. it+2 does work)
- ContiguousIterator guarantees that elements are stored in memory contiguously
  - extends RandomAccessIterator
  - code predating C++17 often treats RandomAccessIterators of std::string, std::vector, and std::array as if they were ContiguousIterators

Outlook: ranges in C++20

- the ranges library (since C++20) enables some functional-style programming on ranges

```cpp
#include <iostream>
#include <ranges>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

    auto results = numbers
                   | std::views::filter([](int n){ return n % 2 == 0; })
                   | std::views::transform([](int n){ return n * 2; });

    for (auto v: results)
        std::cout << v << " "; // outputs: 4 8 12
}
```

Ranges and views

- a range is anything that can be iterated over
  - i.e. it has to provide a `begin()` iterator and an `end()` sentinel
  - containers in the standard library are obviously ranges
- a view is something that you apply to a range and which performs some operation
  - a view does not own data
  - time complexity to copy/move/assign is constant
  - can be composed using the pipe symbol |
  - views are lazily evaluated
  - the standard library provides lots of default views (see reference)

Ranges example

```cpp
#include <iostream>
#include <ranges>

bool isPrime(int i) { // not very efficient, but simple!
    for (int j = 2; j*j <= i; ++j) {
        if (i % j == 0) return false;
    }
    return true;
}

int main() {
    std::cout << "20 prime numbers starting with 1'000'000:\n";
    for (int i : std::views::iota(1'000'000)
                 | std::views::filter(isPrime)
                 | std::views::take(20) ) {
        std::cout << i << " ";
    }

    std::cout << "\n";
}
```

- for more info on the ranges library, check out the reference
    - or the C++20 book of R. Grimm

Contents

Algorithms library

- the algorithms library is another part of the C++ standard library
- it defines common operations on ranges of elements `[first, last)`, such as sorting, searching, manipulating, etc.
- ranges are usually specified using any appropriate iterator type
- spread over 4 headers: `<algorithm>`, `<numeric>`, `<memory>`, `<cstdlib>`
- we will show only a few examples of `<algorithm>`
    - *side note:* for these, the ranges library also has equivalents
        - e.g. `std::sort` and `std::ranges::sort`

std::sort

- 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 (not stable)
- needs $O(n \log n)$ comparisons

```cpp
#include <algorithm>
#include <vector>

int main() {
    std::vector<unsigned> v {3, 4, 1, 2};
    std::sort(v.begin(), v.end()); // 1, 2, 3, 4
}
```

`std::sort` with 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 <

```cpp
#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
}
```

Other sorting operations

- sometimes `std::sort` is not optimal
    - not stable, i.e. does not necessarily keep order of equal-ranked elements
    - sorts the entire range (unnecessary e.g. for top-k queries)
- stable sort:
    - `std::stable_sort`
- partially sorting a range:
    - `std::partial_sort`
- check if a range is sorted:
    - `std::is_sorted` and `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
- arguments against sorting:
  - externally prescribed order that may not be modified
  - frequent updates or insertions
- general semantics:
  - search operations return iterators pointing to result
  - unsuccessful operations are usually indicated by returning the last iterator of a range `[first, last)`

Searching: unsorted

- find the first elements 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`
- and many more... (see reference documentation)

std::find example

```cpp
#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

```cpp
#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 available
  - complexity $O(\log n)$ when range is given as `RandomAccessIterator`
  - can employ custom comparison function (see above)
  - caution: when called with `ForwardIterator` 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 sorted 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

```cpp
#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 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

```cpp
#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`

- initializes 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++()`

```cpp
#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 ascending order)

```cpp
#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 ranged is then sorted in descending 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}
}
```

Further functionality

- the algorithms library offers many more operations, for example:
  - `std::min` and `std::max` over a range instead of two elements
  - `std::merge` and `std::in_place_merge` for merging of sorted ranges
  - multiple set operations (intersection, union, difference, ...)
  - heap functionality using `std::make_heap`, `std::push_heap`, `std::pop_heap` etc.
  - sampling of elements using `std::sample`
  - swapping elements using `std::swap`
  - range modifications:
    - `std::copy` to copy elements to a new location
    - `std::rotate` to rotate a range
    - `std::shuffle` to randomly reorder elements
- for much more: see reference documentation!

Contents

Streams

- the main concept of the I/O library is a stream
- streams are classes organized in a class hierarchy
  - `std::istream` is the base class for input operations, e.g. via `operator>>`
  - `std::ostream` is the base class for output operations, e.g. via `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 represents stdout

- like 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 stream operations

- all streams are subclasses of `std::basic_ios` and support the following 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 << "\n";
else
    std::cout << "error\n";
```

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 characters 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 read and written from a `std::string`
  - defined in 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 via `str()`

```cpp
std::stringstream stream("1 2 3");
int value;
stream >> value; // value == 1

stream.str("4"); // set stream contents to "4"
stream >> value; // value == 4

stream << "foo";
stream << 123;
stream.str(); // == "foo123"
```

File streams

- there are several streams for file I/O in the `<fstream>` header:
    - `std::ifstream`: input file stream to read 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";
```

Caveats of streams

- streams work very well, but sometimes they are not very efficient
  - streams use virtual functions and virtual inheritance which can sometimes be a significant performance overhead
  - streams respect the system's locale settings (e.g. using a period or comma for floating point numbers), making them slow
  - in particular, parsing non-character or non-string types can be quite slow
- if you need more speed, or want more convenience, use libfmt (before C++20) or std::format (since C++20)
  - uses Python-like syntax (see reference)
  - simple example:

```
std::cout << std::format("{} {}: {}!\n", "Hello", "World", 2021);
```

Contents

Multi-Threading

- multiple threads can use the same memory simultaneously
    - multiple threads may read from the same memory location
    - all other accesses (i.e. read-write, write-read, write-write) are conflicts
- conflicting operations are only allowed when threads are synchronized
    - for example using mutexes or atomic operations
- unsynchronized accesses (data races), deadlocks, and other potential issues when using threads are undefined behavior!
- in the following we are showing only a few parts of the C++ multi-threading features

Threads

- the class `std::thread` is defined in header `<thread>`
- it can be used to spawn new threads platform-independently

```cpp
void foo(int a, int b);

// start a thread that calls foo(123, 456)
std::thread t1(foo, 123, 456);

// also works with lambdas
std::thread t2([] { foo(123, 456); } );

// creates an object that does not refer to an actual thread
std::thread t3;
```

Joining threads

- the member function `join()` is used to wait for a thread to finish
  - `join()` must be called exactly once for each thread!
  - if a thread was not joined and the `std::thread` destructor is called, the program is terminated

```
        std::thread t1([] { std::cout << "Hi!\n"; });
        t1.join();

        {
            std::thread t2([] {});
        }
        // program terminated because t2.join() was not called
        // when t2 was destructed
```

- alternatively, you can call `detach()` to have the thread execute independently from its creator
  - though then it can no longer `join()`

std::jthread

- C++20 adds `std::jthread`, a joining thread
  - it automatically joins in its destructor

```
            std::jthread t1([] { std::cout << "Hi!\n"; });
            // automatically joins when going out of scope

            {
                std::jthread t2([] {});
            }
            // OK, t2 joined when going out of scope
```

- you can still explicitly ask it to `join()` or `detach()`, if you want to
- additionally it supports interruption signaling via `std::stop_token`

Thread ownership

- std::thread and std::jthread are not copyable, but they can be moved
    - moving a thread transfers all resources associated with the running thread, and only the moved-to thread can be joined

```
std::thread t1([] { std::cout << "Hi!\n"; });
std::thread t2 = std::move(t1); // t1 now empty
t2.join(); // OK, joining the thread that started in t1 originally
```

- std::thread can be used in containers

```
std::vector<std::thread> threadPool;
for (int i = 1; i < 10; ++i)
    threadPool.emplace_back([i] { safe_print(i); });

for (auto& t : threadPool)
    t.join();
```

Other thread library functions

- there are several more functions related to 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

Cooperative interruption of std::jthread

- std::jthread can be cooperatively interrupted

```cpp
// various #includes omitted to fit slide

using namespace std::literals;

int main() {
    std::jthread interruptible([] (std::stop_token stoken) {
        int counter{0};
        while (true) { // infinite loop
            std::this_thread::sleep_for(0.2s);
            if (stoken.stop_requested()) return;
            std::cout << "interruptible: " << counter << "\n";
            ++counter;
        }
    });

    std::this_thread::sleep_for(1s);

    interruptible.request_stop();
    std::cout << "Main thread interrupted the jthread.\n";
}
```

Contents

Mutual exclusion

- mutual exclusion is a central concept to synchronize threads
- in the headers `<mutex>` and `<shared_mutex>` there are several useful classes for this:
  - `std::mutex` for mutual exclusion
  - `std::recursive_mutex` for recursive mutual exclusion
  - `std::shared_mutex` for mutual exclusion with shared locks
- for locking/unlocking mutexes there are RAII wrappers:
  - `std::unique_lock` for `std::mutex`
  - `std::shared_lock` for `std::shared_mutex`
  - `std::scoped_lock` for deadlock free locking of several mutexes

Mutexes

- `std::mutex` is an exclusive synchronization primitive
- `lock()` locks the mutex, blocking all other threads
    - calling `lock()` will block until the mutex is unlocked
- `unlock()` will unlock the mutex
    - each `lock()` must have a matching `unlock()`!
- `try_lock()` tries to lock the mutex and returns `true` if successful
- all three functions may be called simultaneously by different threads

```
std::mutex printMutex;

void safe_print(int i) {
    printMutex.lock();
    std::cout << i << " ";
    printMutex.unlock();
}
```

Recursive mutexes

- recursive mutexes are regular mutexes that additionally allow a thread that currently holds the mutex to lock it again
  - this is useful for functions that call each other and use the same mutex
- `std::recursive_mutex` implements this, it has the same interface as `std::mutex`
  - `unlock()` still has to be called for each `lock()`!

```cpp
std::recursive_mutex m;

void foo() {
    m.lock();
    std::cout << "foo\n";
    m.unlock();
}

void bar() {
    m.lock();
    std::cout << "bar\n";
    foo(); // this will not deadlock!
    m.unlock();
}
```

Shared mutex

- shared mutexes differentiate between shared and exclusive locks
    - a shared mutex can either be locked exclusively by one thread, or have multiple shared locks
    - this is used for read/write locks, i.e. the readers use shared locks, the writers use exclusive locks
- implemented in `std::shared_mutex`
    - the member functions `lock()` and `unlock()` are for exclusive locks
    - the member functions `lock_shared()` and `unlock_shared()` are for shared locks
    - the member functions `try_lock()` and `try_lock_shared()` try to get an exclusive or shared lock and return `true` on success

Shared mutex: example

```cpp
int value{0};
std::shared_mutex m;
std::vector<std::jthread> threadPool;

// add readers
for (int i = 0; i < 5; ++i)
    threadPool.emplace_back([&] {
        m.lock_shared();
        safe_print(value);
        m.unlock_shared();
    });

// add writers
for (int i = 0; i < 5; ++i)
    threadPool.emplace_back([&] {
        m.lock();
        ++value;
        m.unlock();
    });
```

Working with mutexes

- mutexes requires matching `lock()` and `unlock()` calls from the same thread
  - hence, prefer to use the RAII wrappers (see next slide)
- a thread *A* should not wait for a mutex from thread *B* to be unlocked if *B* needs to lock a mutex that *A* is currently holding (i.e. avoid deadlocks!)
- other caveats to consider when using a mutex in your class:
  - `lock()` and `unlock()` are non-const, hence if const member functions of your class need to use the mutex, you have to make the mutex `mutable`
  - if a member function that locks the mutex calls other member functions, this can lead to deadlocks (use a recursive mutex to avoid this)

`std::unique_lock`: RAII wrapper for mutexes

- `std::unique_lock` wraps mutexes using RAII
  - the constructor calls `lock()`
  - the destructor calls `unlock()`
- `std::unique_lock` can be moved to transfer ownership of the locked mutex

```cpp
std::mutex m;
int i{0};

std::jthread t{[&] {
    std::unique_lock l{m}; // m.lock() is called
    ++i;
    // m.unlock() is called
}};
```

`std::shared_lock`: RAII wrapper for shared mutexes

- shared mutexes also have an RAII wrapper to call `lock_shared()` and `unlock_shared()`
- use `std::shared_lock` for this

```
std::shared_mutex m;
int i{0};

std::jthread t{[&] {
    std::shared_lock l{m}; // m.lock_shared() is called
    std::cout << i << " ";
    // m.unlock_shared() is called
}};
```

Avoiding deadlocks

- when using multiple mutexes, deadlocks can occur
  - in particular when two different threads each succeed to lock a subset of the mutexes and then try to lock the rest
  - consistent ordering avoids this
- use `std::scoped_lock` when locking multiple mutexes in consistent order, to guarantee no deadlocks

Avoiding deadlocks: example

- calling threadA and threadB concurrently can lead to deadlocks:

```
std::mutex m1, m2, m3;
void threadA() {
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    std::unique_lock l3{m3}, l2{m2}, l1{m1};
}
```

- using scoped_lock will lead to no deadlock:

```
std::mutex m1, m2, m3;
void threadA() {
    std::scoped_lock l{m1, m2, m3};
}
void threadB() {
    std::scoped_lock l{m3, m2, m1};
}
```

Contents

Condition variables

- condition variables can be used to synchronize threads by waiting until an (arbitrary) condition becomes true
  - a condition variable uses mutexes to synchronize threads
  - threads can wait on or notify the condition variable

  - when a thread waits on a 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 (cont.)

- `std::condition_variable` is defined in the `<condition_variable>` header
- 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 variable: example

- use case: worker queues, inserting tasks and worker threads are notified to do the task

```cpp
std::mutex m;
std::condition_variable cv;
std::vector<int> taskQueue;

void pushWork(int task) {
    {
        std::unique_lock lck{m};
        taskQueue.push_back(task);
    }
    cv.notify_one();
}
```

```cpp
void workerThread() {
    std::unique_lock lck{m};
    while (true) {
        if (!taskQueue.empty()) {
            int task = taskQueue.back();
            taskQueue.pop_back();
            lck.unlock();
            // ... do actual task here
            lck.lock();
        }
        cv.wait(lck,
            []{ return !taskQueue.empty(); });
    }
}
```

- note: `wait()` should always use a condition to avoid deadlock

Contents

Atomic operations

- threads can also be synchronized with atomic operations that are usually more efficient than mutexes
  - atomic means that an operation is executed as one unit, no other operation on the same object can be executed at the same time
  - note: only the atomicity of single operations are guaranteed to be atomic
- example, assuming that `void atomic_add(int& p, int i)` represents an atomic operation that adds `i` to `p`

```
int a{10};
void threadA() { atomic_add(a, 1); }
void threadB() { atomic_add(a, 2); }
```

  - when `threadA()` and `threadB()` are called concurrently, it is always guaranteed `a == 13` afterwards

Atomic operations library

- the atomic operations library in `<atomic>` provides several classes and functions
- `std::atomic<T>` is a class that represents an atomic version of the type `T`
  - `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
  - `compare_exchange_weak(...)` and `compare_exchange_strong(...)` perform compare-and-swap (see reference)
- if `T` is an integral type, the following operations also exist:
  - `T fetch_add(T arg)`: adds `arg` to the value and returns the old value
    - same for `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`
  - `T operator+=(T arg)`: add `arg` to the value and returns the new value
    - same for `operator-=`, `operator&=`, `operator|=`, `operator^=`
  - `operator++` and `operator--` for increment/decrement

Atomic operations example

- the modifications of a single atomic object are totally ordered in the modification order
    - the modification order is consistent between threads, i.e. all threads see the same order
    - the modification order is only total for individual atomic objects

```
std::atomic<int> i{0};
void workerThread() {
    i += 1; // (A)
    i -= 1; // (B)
}
void readerThread() {
    int iLocal = i.load();
    assert(iLocal == 0 || iLocal == 1); // always true!
}
```

- because the memory order is consistent between threads, the reader thread will never see an execution order of, for example, (A), (B), (B), (A)

Atomic references

- `std::atomic` will not work as expected with references
  - it will create a copy on initialization
- C++20 introduced `std::atomic_ref` for use with references:

```cpp
int val{5};
int& ref = val;
std::atomic<int> notReallyRef(ref);
std::atomic_ref<int> atomicRef(ref);
++notReallyRef; // will increment only the internal copy of ref
++atomicRef;    // will actually increment the referenced variable
++atomicRef;

std::cout << "ref: " << ref << "\n";
std::cout << "notReallyRef: " << notReallyRef.load() << "\n";
std::cout << "atomicRef: " << atomicRef.load() << "\n";
```

Atomic smart pointers

- `std::shared_ptr` guarantees atomic operations on the internal reference counter, but not for the managed resource
- C++20 introduced a specialization of `std::atomic` for `std::shared_ptr` and `std::weak_ptr`

Atomic flags

- `std::atomic_flag` provides an atomic boolean with a special interface
- it is guaranteed to be lock-free
- it can be set to:
    - `true` by calling `test_and_set()`
    - `false` by calling `clear()`
- `test()` returns the current value
- it additionally provides `wait(bool)`, `notify_one()`, and `notify_all()`, similar to a condition variable

Contents

Outlook on C++20 further facilities

- C++20 introduced many more parallel programming features
- examples include:
  - coroutines, i.e. functions that can suspend and resume their execution while keeping their state
  - counting semaphores for shared resources
  - latches and barriers to coordinate threads
  - cooperative interruption via `std::stop_token`, `std::stop_callback`, `std::stop_source`
  - synchronized output streams via `std::osyncstream`

Contents

C++20 concepts

- concepts allow
  - specifying requirements for template parameters
  - overloading of functions
  - specialization of class templates
- concepts can be used in
  - function templates
  - class templates
  - generic member functions of classes / class templates
- instead of `auto` you can also use a concept
- use pre-defined concepts of the standard library, or define your own

Credit for this chapter goes to R. Grimm's book: *C++20*

Using concepts

- with a `requires` clause:

```
template <typename T>
requires std::integral<T>
auto gcd1(T a, T b) { /* ... */ }
```

- with a trailing `requires` clause:

```
template <typename T>
auto gcd2(T a, T b) requires std::integral<T>
{ /* ... */ }
```

- using a constrained template parameter:

```
template <std::integral T>
auto gcd3(T a, T b) { /* ... */ }
```

- using an abbreviated function template:

```
auto gcd4(std::integral auto a, std::integral auto b) { /* ... */ }
```

`requires` clauses

- `requires` clauses specify
  - a constraint on template parameters
  - or a constraint on a function declaration
- the constraint after `requires` must be a compile-time predicate, such as
  - a named concept (from the standard library or defined by you)
  - a conjunction/disjunction of named concepts
  - a requires expression (see later)

Concepts as return types

- you can use concepts as return types of functions:

```
// with requires clause:
template <typename T>
requires std::integral<T>
std::integral auto gcd1(T a, T b) { /* ... */}

// abbreviated function template:
std::integral auto gcd4(std::integral auto a, std::integral auto b)
{ /* ... */ }
```

- basically anywhere you can put `auto`, you can put `<concept> auto` instead

Concepts: example

```
#include <concepts>
#include <iostream>

struct NotCopyable {
    NotCopyable() = default;
    NotCopyable(const NotCopyable&) = delete;
};

template <typename T>
struct MyVector {
    void push_back(const T&) requires std::copyable<T> {}
};

int main() {
    MyVector<int> myVec1;
    myVec1.push_back(2021); // OK

    MyVector<NotCopyable> myVec2;
    myVec2.push_back(NotCopyable()); // ERROR: not copyable
}
```

Concepts: example with variadic templates

```cpp
// not listing #includes for space reasons

template <std::integral... Args>
bool all(Args... args) { return (... && args); }

template <std::integral... Args>
bool any(Args... args) { return (... || args); }

template <std::integral... Args>
bool none(Args... args) { return not(... || args); }

int main() {
    std::cout << std::boolalpha << "\n"; // enable bool as text output
    std::cout << "all(5, true, false): "
              << all(5, true, false) << "\n";  // outputs "false"

    std::cout << "any(5, true, false): "
              << any(5, true, false) << "\n";  // outputs "true"

    std::cout << "none(5, true, false): "
              << none(5, true, false) << "\n"; // outputs "false"
}
```

Overloading function templates on concepts

```cpp
// not listing #includes for space reasons

template<std::forward_iterator I>
void advance(I& iter, int n) { std::cout << "forward_iterator\n"; }

template<std::bidirectional_iterator I>
void advance(I& iter, int n) { std::cout << "bidirectional_iterator\n"; }

template<std::random_access_iterator I>
void advance(I& iter, int n) { std::cout << "random_access_iterator\n"; }

int main() {
    std::forward_list fwdList{1, 2, 3}; auto flIter = fwdList.begin();
    advance(flIter, 2); // outputs: forward_iterator

    std::list list{1, 2, 3}; auto listIter = list.begin();
    advance(listIter, 2); // outputs: bidirectional_iterator

    std::vector vector{1, 2, 3}; auto vecIter = vector.begin();
    advance(vecIter, 2); // outputs: random_access_iterator
}
```

Specializing templates on concepts

```cpp
#include <concepts>
#include <iostream>

template <typename T>
struct Vector {
    Vector() { std::cout << "Vector<T>\n"; }
};

template <std::regular T>
struct Vector {
    Vector() { std::cout << "Vector<std::regular>\n"; }
};

int main() {
    Vector<int> myVec1;  // outputs: Vector<std::regular>
    Vector<int&> myVec2; // outputs: Vector<T>
}
```

- semiregular means copyable, movable, swappable, and default constructible
- regular means semiregular and equality comparable

Using more than one concept

- more than one concept can be used at the same time
- example:

```
template <typename Iter, typename Val>
    requires std::input_iterator<Iter>
          && std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v);
```

- you can also mix and match styles:

```
template<std::input_iterator Iter, typename Val>
    requires std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v);
```

Overloading abbreviated function templates

```cpp
#include <concepts>
#include <iostream>

void overload(auto t) {
    std::cout << "auto:" << t << "\n";
}

void overload(std::integral auto t) {
    std::cout << "integral: " << t << "\n";
}

void overload(long t) {
    std::cout << "long: " << t << "\n";
}

int main() {
    overload(3.14);     // outputs: auto: 3.14
    overload(2011);     // outputs: integral: 2011
    overload(2021L);    // outputs: long: 2021
}
```

Selected pre-defined concepts

- here are a few pre-defined concepts from the concepts library `<concepts>` (for details see reference)
  - `same_as`, `derived_from`, `convertible_to`, `assignable_from`, `swappable`
  - `integral`, `signed_integral`, `unsigned_integral`, `floating_point`
  - `destructible`, `constructible_from`, `default_constructible`, `move_constructible`, `copy_constructible`
  - `equality_comparable`, `totally_ordered`
  - `movable`, `copyable`, `semi_regular`, `regular`
- and many more...

Defining your own concepts

- syntax for defining your own concept:

```
template <template-parameter-list>
concept concept-name = constraint-expression;
```

- a constraint expression is either a
  - logical combination of other concepts or compile-time predicates (i.e. something that returns a `bool` at compile-time)
  - or a requires expression, i.e.
    - simple requirement
    - type requirement
    - compound requirement
    - nested requirement

Constraint expression example

- example for a logical combination of other concepts or compile-time predicates:

```cpp
#include <type_traits>

template <typename T>
concept Integral = std::is_integral_v<T>;

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;

template <typename T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

- this is equivalent to the concepts `integral`, `signed_integral` etc. from `<concepts>`

Requires expressions

A requires expression can be

- a simple requirement:

```cpp
template <typename T>
concept Addable = requires (T a, T b) {
    a + b;
};
```

- a type requirement:

```cpp
template <typename T>
concept TypeRequirement = requires {
    typename T::value_type; // T has nested member value_type
    typename Other<T>;      // Other can be instantiated with T
};
```

- (continued on next slide)

Requires expressions (cont.)

- a compound requirement:

```
template <typename T>
concept Equal = requires (T a, T b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};
```

- a nested requirement:

```
template <typename T>
concept UnsignedIntegral = Integral<T> &&
requires (T) {
    requires !SignedIntegral<T>;
};
```

Example concept

```
template<typename I> concept random_access_iterator =
    bidirectional_iterator<I> &&
    derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
    totally_ordered<I> &&
    sized_sentinel_for<I, I> &&
    requires (I i, const I j, const iter_difference_t<I> n) {
        { i += n } -> same_as<I&>;
        { j + n } -> same_as<I>;
        { n + j } -> same_as<I>;
        { i -= n } -> same_as<I&>;
        { j - n } -> same_as<I>;
        { j[n] } -> same_as<iter_reference_t<I>>;
    };
```

- using e.g. `std::sort` on a `std::list` will now produce a readable error message!

Contents

History of C++ standards

Early C++:

- C++98: the first ISO standard for C++
- C++03: minor revision

Modern C++:

- C++11: major revision
- C++14: minor revision
- C++17: medium revision
- C++20: major revision

Development has settled on a 3 year cadence.

C++20

- C++20:
  - finalized in February 2020
  - feature-freeze already in July 2019
  - approved in September 2020
  - published by ISO in December 2020
- it was a major revision
  - a few new big features (like concepts, coroutines, ranges)
  - a major disruptor (modules)
  - and some smaller things (format, spaceship, constexpr etc.)

C++20: adoption

- due to the breaking nature of modules, it is expected to take a long time for full support throughout the tool chains
- the other C++20 language features are mostly available in the recent versions of the major compilers
- quite a few of the C++20 standard library features will take a little longer to be supported in all the major compilers

Contents

Future C++ standards: what's on the horizon?

- C++23 is supposed to contain
  - library support for coroutines (such as tasks and generators)
  - a modular standard library (using C++20 modules)
    - MSVC suggests, for example: `import std.core;`
  - executors
    - rules about where, when, and how to run a callable
  - network library
    - support e.g. for TCP, UDP, DNS, IPv4/v6, client/server applications
    - but not for network protocols (HTTP, SMTP, SSL etc.)

(see also `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r3.html`)

Future C++ standards: what's on the horizon? (cont.)

- beyond C++23, plans exist for
  - contracts
    - to define pre-conditions, post-conditions, and assertions
  - compile-time reflection
    - to acquire meta information on types, aliases, members, objects etc.
  - pattern matching
    - `inspect` to allow "switch" on `std::tuple`, `std::variant`, polymorphic types etc.

C++23: what's already in there

- quite a few (mostly minor) things are already approved for inclusion in the C++23 draft, for example:
  - literal suffixes for `std::size_t`
  - a stracktrace library (based on Boost.Stacktrace)
  - `consteval if`
  - more `constexpr` possibilities
  - improvements to the ranges library
  - improvements to the format library
  - and many more

What can you do?

- try out the C++20 and experimental C++23 features!
  - e.g. using the Compiler Explorer (`https://godbolt.org`)
  - e.g. using Wandbox (`https://wandbox.org`)

- do a student project using C++
  - blatant plug: check out `https://ciip.in.tum.de/projects.html`
  - e.g. projects involving our open-source C++ library elsa: `https://gitlab.lrz.de/IP/elsa`
  - there are of course other groups using C++ heavily, like the database folks

Thank you!