Concepts of C++ Programming

Alexis Engelke

Winter 2024/25

Contents

1. Overview and Hello World

1.1. Organization

[Slide 2] Module "Concepts of C++ Programming" (CIT323000)

Goals

- Write good and modern C++ code
- Apply widely relevant C++ constructs
- Understand some advanced language concepts

Non-Goals

- Become experts in C++
- Fancy language features
- Apply involved optimizations

Prerequisites

- Fundamentals of object-oriented programming — EIDI, PGdP
- Fundamentals of data structures and algorithms — GAD
- Beneficial: operating systems, computer architecture — GBS, ERA

> This lecture assumes knowledge of imperative and object-oriented programming languages like Java (e.g., `for` loops, classes, visibility, inheritance, polymorphism).

[Slide 3] Lecture Organization

- Lecture: Mon 14:30 – 17:00, MW 0001
  - Lecturer: Dr. Alexis Engelke `engelke@in.tum.de`
  - Live stream and recording via RBG: `https://live.rbg.tum.de/`
  - Tweedback for questions during lecture
- Exercises: Tue 14:15 – 15:45, Interims II HS 3
  - Florian Drescher, Mateusz Gienieczko
- Material: `https://db.in.tum.de/teaching/ws2425/cpp/`
- Zulip-Streams: #CPP, #CPP Homeworks, #CPP Random/Memes
- Exam: written exam *on your laptop*, on-site, 90 minutes
  - Open book, but no communication/AI tools allowed
  - Same submission system as for homework

[Slide 4] Homework

- 1–2 programming tasks as homework every week

  - Released on Monday, deadline next Sunday 11:59 PM
- Automatic tests and grading, points only for completely solved tasks
  - Typically all[1] tests provided with the assignment
- Container environment provided, no support for other setups
- Submission via git+ssh only
- Grade bonus: 0.3 for 75% of exercise points
  - Applies **only** for the main exam, not for the retake
- Cheating in homework ⇝ 5.0U in final grade

[Slide 5] Literature

Primary

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

Supplementary

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

1.2. Introduction

[Slide 6] What is C++?

- Multi-paradigm general-purpose programming language
  - Imperative programming
  - Object-oriented programming
  - Generic programming
  - Functional programming
- Key characteristics
  - Compiled
  - Statically typed
  - Facilities for low-level programming

[Slide 7] Some C++ History

Initial development

- Bjarne Stroustrup at Bell Labs (since 1979)

[1] We may add extra cases to prevent hard-coding of test cases.

  - Originally "C with classes", renamed in 1983 to C++
- In large parts based on C
- Inspirations from Simula67 (classes) and Algol68 (operator overloading)
- Initially developed as a C++-to-C converter (Cfront)

First ISO standardization in 1998 (C++98)

- Further amendments in following years (C++03/11/14/17/20)
- Current standard: C++23

[Slide 8] C++ Standard vs. Implementations

- C++ *standard* specifies requirements for C++ *implementations* about language features and standard library
- "Implementation" consists of: compiler, standard library impl, OS, ...
- Some things are specified rigidly in the standard
- Some things are *implementation-defined*
  - Standard specifies options, implementation chooses one and documents that
  - Example: size of an `int`
- Implementations can offer extensions[2]

> Typically, implementations of the C++ compiler, the C++ standard library, the underlying C standard library, and the operating system are separated. Obviously, only few combinations are supported, but some compilers like Clang support using different standard library implementations (e.g. with `-stdlib=libc++`).
>
> - Popular C++ compilers: Clang, GCC, MSVC, EDG eccp (used as foundation for some other commercial compilers)
> - Popular C++ standard library implementations: libstdc++ (GNU), libc++ (Clang/LLVM project), MSVC STL (Microsoft)
> - Popular C standard library implementations: glibc (GNU), Microsoft CRT, (musl, does not support C++)

[Slide 9] Why Study C++?

- Performance
  - Very flexible level of abstraction
  - Direct mapping to hardware capabilities easily possible
  - Zero-overhead rule: "What you don't use, you don't pay for."
- Scales to large systems (with some discipline)
- Interoperability with other languages, esp. C
- *Huge* amount of legacy code needs developers/maintainers
  - compilers, databases, simulations, ...

[2] `https://clang.llvm.org/docs/LanguageExtensions.html`

> Studying C++ does not preclude studying other languages. C++ is not the best or right tool for every job, so you probably want to learn at least half a dozen programming languages. (My personal picks in 2024 are: C, C++, Python, Rust, Go, JavaScript.)
>
> Note that there's no such thing as a "zero-cost abstraction". They do have cost, typically during compilation. Some of these "zero-cost" abstractions *also* have some run-time cost. For example, the mere possibility of C++ exceptions can prevent optimizations.

[Slide 10] This Lecture

- Go bottom-up through important language constructs
  - Some things (e.g. standard library) appear rather late
  - Cyclic dependencies are unavoidable
- Focus: widely used constructs and important cases
  - Topic selection based on relevance real-world projects
  - Many special cases not discussed, lecture will be inaccurate at times
  - Use the C++ reference!

1.3. Hello World!

[Slide 11] Hello World!

```
#include <print>
int main() {
  std::println("Hello␣World!");
  return 0;
}
```

On the command line:

```
$ clang++ -std=c++23 -o hello hello.cpp
$ ./hello
Hello World!
```

[Slide 12] Hello World, explained[3]

```
// Make print and println available
#include <print>

// Definition of function main().
// Program execution starts at main.
int main() {
  // std:: is a namespace prefix. std is for the C++ standard library
  std::println("Hello␣World!");

  // End program with exit code 0. (zero = everything ok, non-zero = error)
  return 0;
}
```

[3] A bit hand-wavy, but we have to start somewhere.

[Slide 13] Program Arguments

- `main` can take two paramters to hold command-line arguments
  - `int argc`: number of arguments
  - `char** argv`: the actual arguments, ∼array of strings
  - First argument is the program invocation itself (e.g., `./hello2`)

```
#include <print>
int main(int argc, char** argv) {
  std::println("Hello␣{}!", argv[1]); // DON'T DO THIS
  return 0;
}
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Segmentation fault
```

> The program crashed! A "segmentation fault" is an access to an invalid memory address that was caught by the operating system. In this case, we accessed the second element of an array of size 1 (argc is 1). We were **lucky**[a] and got a crash! Since there are no bounds checks, something completely different could have happened.
>
> In this example, this might be easy to see, but we will very briefly look at two important debugging strategies.
>
> [a] Ok, this example will always crash. But regardless, never rely on getting crashes on mistakes.

[Slide 14] Debugging 101

- Pass `-g` to Clang to enable debug info generation
- Run `gdb ./hello2`

```
$ clang++ -g -std=c++23 -o hello2 hello2.cpp
$ gdb ./hello2
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
(gdb) backtrace
// ...
#16 in main (argc=0x1, argv=0x7fffffffe868) at hello2.cpp:3
(gdb) up 16
(gdb) print argc
1
(gdb) quit
```

> To start debugging run the command `gdb myprogram`. This starts a command-line interface. Here are some useful commands:

`help`	Show general help or help about a command.
`run`	Start the debugged program.
`break`	Set a breakpoint. When the breakpoint is reached, the debugger stops the program and accepts new commands.
`delete`	Remove a breakpoint.
`continue`	Continue running the program after it stopped at a breakpoint or by pressing Ctrl+C.
`next`	Continue running the program until the next source line of the current function.
`step`	Continue running the program until the source line changes.
`nexti`	Continue running the program until the next instruction of the current function.
`stepi`	Execute the next instrution.
`print`	Print the value of a variable, expression or CPU register.
`frame`	Show the currently selected *stack frame*, i.e. the current stack with its local variables. Usually includes the function name and the current source line. Can also be used to switch to another frame.
`backtrace`	Show all stack frames.
`up`	Select the frame from the next higher function.
`down`	Select the frame from the next lower function.
`watch`	Set a watchpoint. When the memory address that is watched is read or written, the debugger stops.
`thread`	Show the currently selected thread in a multi-threaded program. Can also be used to switch to another thread.

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

The documentation for `gdb` can be found here: `https://sourceware.org/gdb/current/onlinedocs/gdb/`

[Slide 15] Debugging 102

- Print debugging.

```
#include <print>
int main(int argc, char** argv) {
  std::println("argc={}", argc);
  std::println("Hello␣{}!", argv[1]);
  return 0;
}
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Segmentation fault
```

> Print debugging is an extremely simple, but also an extremely powerful technique. I personally use print debugging most of the time and only resort to debuggers like GDB in complex situations.

[Slide 16] Program Arguments, attempt 2

```
#include <print>
int main(int argc, char** argv) {
  if (argc >= 2)
    std::println("Hello␣{}!", argv[1]);
  else
    std::println("Hi␣there!");
  return 0;
}
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Hi there!
```

[Slide 17] Compiler Flags

Compiler invocation: `clang++ [flags] -o output inputs...`

- `-std=c++23` — set standard to C++23
  - Always specify the version of the C++ standard!
- `-g` — enable debugging information
- `-Wall` — enable many warnings
- `-Wextra` — enable some more warnings
  - Always compile with `-Wall -Wextra`! Warnings often hint at bugs.
- `-O0` — no optimization, typically good for debugging
- `-O1`/`-O2`/`-O3` — enable optimizations at specified level

1.4. CMake

[Slide 18] Build Systems: CMake

- Frequent use of long compiler commands is tedious and error-prone
- Manual work doesn't scale to larger projects
- Different systems may require different flags
- CMake: build system specialized for C/C++
  - Widely used by large projects and supported by many IDEs
- `CMakeLists.txt` specifies project, files, etc.
- Reference: `https://cmake.org/cmake/help/latest/`

[Slide 19] CMake Example

CMakeLists.txt:

```
# Require a specific CMake version, here 3.20 for C++23 support
cmake_minimum_required(VERSION 3.20)
# Set project name, required for every project
project(hello2)
# We use C++23, basically adds -std=c++23 to compiler flags
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
# Compile executable hello2 from hello2.cpp
add_executable(hello2 hello2.cpp)
```

On the command line:

```
$ mkdir build; cd build # create separate build directory
$ cmake ..
$ cmake --build .
$ ./hello2
```

[Slide 20] Further CMake Commands and Variables

- `add_executable(myprogram a.cpp b.cpp)`
  Define an executable to be built from the source files `a.cpp` and `b.cpp`
- `add_compile_options(-Wall -Wextra)`
  Add `-Wall -Wextra` to compiler flags
- `set(CMAKE_CXX_COMPILER clang++)`
  Set C++ compiler to `clang++`
- `set(CMAKE_BUILD_TYPE Debug)`
  Set "build type" `Debug` (other values: `Release`, `RelWithDebInfo`); affects optimization and debug info

Variables can be set on the command line invocation of CMake:

```
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

[Slide 21] Overview and Hello World – Summary

- C++ is a compiled, widely-used, multi-paradigm language
- Program execution typically starts at `int main()`
- Command line arguments accessible via `argc`/`argv`
- Basic debugging techniques: GDB and print debugging
- Important compiler options for warnings and optimizations
- Basic usage of CMake for building C++ projects

[Slide 22] Overview and Hello World – Questions

- What are key characteristics of the C++ language?
- Why is C++ one of the most important languages today?
- How to access program arguments?
- What are important flags for compiling C++ code with Clang?
- How to debug a compiled C++ program with GDB?
- What is a segmentation fault?
- What are advantages of using a build system like CMake?

2. Basic Syntax and Object Model

[Slide 24] Reminder: C++ Reference

These slides will necessarily be inaccurate or incomplete at times. Use the reference! `https://en.cppreference.com/w/cpp`

[Slide 25] Comments[1]

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

Example:

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

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

2.1. Types

[Slide 26] Fundamental Types[2]

- `void` – empty type, has no values
  - E.g., used to indicate functions that return no value
- Integer types
  - Boolean type: `bool` (1-bit integer, `true`/`false`)
  - Integer types: `int`, `long`, `unsigned long`, ...
  - Character types: `char`, `char16_t`, ...
- Floating-point types
  - `float`, `double`, `long double`

[Slide 27] Integer Types

- Sign modifiers: `signed` (default), `unsigned`
- Size modifiers: `short`, `long` (≥32 bit), `long long` (≥64 bit)

[1] `https://en.cppreference.com/w/cpp/comment`
[2] `https://en.cppreference.com/w/cpp/language/types`

- Keyword: `int` (optional if modifiers are present)
- Order of keywords is arbitrary
  - `unsigned long long` = `long unsigned int long`
- Signed integers use two's complement (since C++20)

> By convention, sign modifiers come first, `signed` is omitted, and `int` comes last, but is omitted if a size modifier is present.

[Slide 28] Integer Types: Minimum Width

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

- Exact width of integer types is **not** specified by the standard!

[Slide 29] Fixed-Width Integer Types[3]

- Use fixed-width types from when... a fixed width is required
- `#include <cstdint>`
- `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- But: optional, only available if supported by implementation

- Guideline: use fixed-width types only when really required
  - E.g., data structures where size is important, bitwise operations
  - Otherwise, prefer regular integers

> Don't prematurely "optimize" by using small data types, e.g. in data structures. Modern CPUs are optimized for 32/64-bit arithmetic, and some operations on smaller data types can even be *less* efficient.
>
> There are also `size_t` and `ptrdiff_t`, which are described in when introducing pointers later.

[3] `https://en.cppreference.com/w/cpp/types/integer`

[Slide 30] Integer Literals[4]

- Decimal (`42`), octal (`052`), hexadecimal (`0x2a`), binary (`0b101010`)
- `unsigned` suffix: `42u` or `42U`
- `long` suffix: `42l` or `42L`; `long long` suffix: `42ll` or `42LL`
- Both suffixes can be combined, e.g. `42ul`, `42ull`
- Separable by single quotes, e.g. `1'000'000'000ull`, `0b0010'1010`

Quiz: What is the type of the integer literal `0xdeadcabel`?

(Assume 32-bit `int`, 32-bit `long`, as on, e.g., Windows)

A. `int` B. `long` C. `unsigned long` D. `long long`

> Fun fact: `0` is technically an octal number.

[Slide 31] Character Types

- Represent character codes and integers
- `signed char`, `unsigned char`
- `char` — implementation-defined whether signed/unsigned!
  - Use `char` only for actual characters, not for arithmetic
- Size: defined as 1 byte
- Size of byte: **at least** 8 bit[5]
- For UTF characters: `char8_t` (C++20), `char16_t`, `char32_t`

> The signedness of `char` is platform-dependent. On x86, which always had an instruction for sign extension (`movsx`), `char` tends to be signed. Early ARM processors, in contrast, did not have an instruction for sign extension, so loading a `signed char` from memory required three instructions (load, shift left, arithmetic shift right). To improve efficiency, it was decided to make `char` an unsigned data type. When porting software from x86 to ARM, this occasionally causes problems in practice.
>
> Note that on modern CPUs, the performance difference is very low. Unsigned data types tend to be slightly more efficient in some cases, but this difference is often negligible.
>
> Although a `char` is one byte large, the size of one byte is not specified by the C++ standard and only required to be at least 8 bits. Platforms that use non-8-bit bytes have become increasingly rare over the past decades, but still exist, e.g. some digital signal processors (DSPs). (Note that these tend to have no compilers for modern C++

[4] `https://en.cppreference.com/w/cpp/language/integer_literal`
[5] Might change for C++26 to exactly 8 bits; proposal: `https://wg21.link/p3477r0`

versions. In practice, programs are almost never tested on platforms where `char` is not 8 bits.)

[Slide 32] Character Literals[6]

- E.g. `'a'`, `'b'`, `'€'`
  - Any character from the source character set except: `'`, \, newline
- Escape sequences, e.g. `'\''`, `'\\'`, `'\n'`, `'\u1234'`

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

[Slide 33] Floating-Point Types

- `float` – usually IEEE-754 32-bit binary format
- `double` – usually IEEE-754 64-bit binary format
- `long double` – extended precision, format varies strongly
  - Some platforms use 64-bit (like `double`), e.g. MSVC on x86
  - Some platforms use 128-bit, e.g. usually AArch64 (this is typically a softfloat implementation $\rightsquigarrow$ slow)
  - On x86, typically 80-bit x87 binary floating-point
- Usual caveats of FP arithmetic apply: infinity, signed zero, NaN

Due to the often lower performance and strongly varying accuracy, `long double` is typically only used when the target platform is known and the extra accuracy is needed.

[Slide 34] Floating-Point Literals[7]

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

- `float` suffix: `1.0f` or `1.0F`
- `long double` suffix: `42.0l` or `42.0L`

- Separable by single quotes, e.g. `1'000.000'001`, `.141'592e12`

[6] `https://en.cppreference.com/w/cpp/language/character_literal`
[7] `https://en.cppreference.com/w/cpp/language/floating_literal`

2.2. Operators

[Slide 35] Operator Precedence Table (1)[8]

Prec.	Operator	Description	Associativity
1	`::`	Scope resolution	left-to-right

2	`a++ a--`
	`<type>() <type>{}`
	`a() a[]`
	`. ->`
		Postfix increment/decrement
		Functional Cast
		Function Call/Subscript
		Member Access
			left-to-right

3	`++a --a`
	`+a -a !a ~a`
	`(<type>)`
	`*a &a`
	`sizeof`
	`new new[]`
	`delete delete[]`
		Prefix increment/decrement
		plus/minus/logical not/bitwise not
		C-style cast
		Dereference/Address-of
		Size-of
		Dynamic memory allocation
		Dynamic memory deallocation
			right-to-left

[Slide 36] Operator Precedence Table (2)

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

[8] `https://en.cppreference.com/w/cpp/language/operator_precedence`

[Slide 37] Operator Precedence Table (3)

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

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

17	`,`	Comma	left-to-right

C++ has a wide range of operators with "typical" semantics and mostly typical precedence and associativity. Some operators like the comma operator are rarely used. The left-hand side of an assignment can not only be a variable, but everything that refers to the identity of an object. This will be covered in more detail when discussing value types and references.

Note that even if parenthesis can be omitted, it is sometimes useful to use them anyway for clarity:

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

2.3. Observable Behavior

[Slide 38] Observable Behavior

Observable behavior of C++ programs precisely defined, unless:

- *implementation-defined behavior* – documented by C++ implementation
- *unspecified behavior* – one of multiple options can happen
  - E.g., evaluation order of function arguments: one permutation must happen
- program *ill-formed* – syntax/semantic error, compiler must diagnose
- program *ill-formed, no diagnostic required* – semantically invalid, hard to diagnose
  - Typically not detectable during compilation, not too many cases
- *undefined behavior* – the standard imposes no requirements

[Slide 39] Undefined Behavior[9] (UB)

- Some violations of language rules are undefined behavior: standard enforces no restrictions $\rightsquigarrow$ **anything** can happen
  - Typically cases, where checks would be costly or impossible

$\Rightarrow$ A C++ program **must never** contain undefined behavior!

- Examples: out-of-bounds array access, signed integer overflow, shift by negative index, shift larger than value size, ...
  - Signed integers: UB on overflow; unsigned integers: well-defined wrap
- Compiler can assume that program contains no undefined behavior[10]
  - Allows for more optimizations, e.g. eliminate some checks

[Slide 40] Undefined Behavior – Example

Quiz: Which answer is correct?

```
bool f1(int x) { return x + 1 > x; }
bool f2(unsigned x) { return x + 1 > x; }
```

A. The return value of `f1` is always `false`.
B. The return value of `f2` is always `true`.
C. The return value of `f1` depends on the parameter.
D. The return value of `f2` depends on the parameter.
E. `f2` might invoke undefined behavior.

> Compilers regularly make use of the assumption that undefined behavior doesn't occur. Compile these functions with optimizations and look at the generated assembly code.

2.4. Basic Syntax

[Slide 41] Variables[11]

- Declaration: type specifier followed by declarators (variable names)
- Declarator can optionally be followed by an initializer
- No initializer: *default-initialized*
  - Non-local variables: zero-initialized
  - Local variables: **not initialized**
- Access of uninitialized variable is **undefined behavior**

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

[9] https://en.cppreference.com/w/cpp/language/ub
[10] https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
[11] https://en.cppreference.com/w/cpp/language/declarations

```
  unsigned meaningOfLife = 42;
}
```

> As the type partly written in the type specifier and partly in the declarator (e.g., pointers, references), declaring multiple variables in a single statement is typically considered as error-prone and therefore avoided.

[Slide 42] Variable Initializers[12]

- `variableName(<expression>)`
- `variableName = <expression>`
- `variableName{<expression>}` (error on possible information loss)

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

[Slide 43] Simple Statements[13]

Declaration statement: Declaration followed by a semicolon

```
int i = 0;
```

Expression statement: Any expression followed by a semicolon

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

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

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

[Slide 44] Scope[14]

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

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

```
int a = 21;
int b = 0;
{
  int a = 1; // scope of the first a is interrupted
  int c = 2;
  b = a + c + 39; // a refers to the second a, b == 42
} // scope of the second a and c ends
b = a; // a refers to the first a, b == 21
```

[12] https://en.cppreference.com/w/cpp/language/initialization
[13] https://en.cppreference.com/w/cpp/language/statements
[14] https://en.cppreference.com/w/cpp/language/scope

```
b += c; // ERROR: c is not in scope
```

[Slide 45] If Statement[15]

- Conditionally execute another statement
- Condition converted to `bool` decides which branch is taken
- Optional initialization statement
- Optional `else` branch

```
if (value < 42)
  valueLessThan42();
else
  valueTooLarge();


if (unsigned n = compute(); n > 4) {
  // do something
}
// The latter is equivalent to:
{
  unsigned n = compute();
  if (n > 4) {
    // do something
  }
}
```

> The condition can *also* be a simple declaration, in which case the condition is whether the assigned variable converted to `bool` is true. Examples:
>
> ```
> if (unsigned n = compute()) {
>   // do something if n != 0
> }
> if (unsigned a = compute(0); unsigned b = compute(a)) {
>   // do something if b != 0
> } else {
>   // can also use a and b here
> }
> ```

[Slide 46] If Statement Nesting

- `else` is associated with the closest `if` that has no `else`

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

- When in doubt, use curly braces to make scopes explicit

```
// Working as intended
if (condition0) {
```

[15] https://en.cppreference.com/w/cpp/language/if

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

[Slide 47] Switch Statements[16]

- Conditional control flow transfer based on integral type
- Constant values for `case`, must be unique
- `break` exits `switch`
- Implicit fallthrough!
  - Use `[[fallthrough]];` when intended
- Condition can have declaration

```
switch (compute()) {
case 42:
  // do something for 42
  break;
case 20:
  // do something for 20
  [[fallthrough]];
case 21:
case 22:
  // do something for 20/21/22
  break;
default:
  break;
}
```

[Slide 48] While and Do-While Loops

- `while`:[17] repeatedly execute statement while condition is true
  ```
  unsigned i = 42;
  while (i < 42) {
    // never executed
  }
  ```
- `do`–`while`:[18] like `while`, but execute body at least once
  ```
  unsigned i = 42;
  do {
    // executed once
  } while (i < 42);
  ```
- `break`/`continue` to exit loop/skip remainder of body

[Slide 49] For Loops[19]

[16] https://en.cppreference.com/w/cpp/language/switch
[17] https://en.cppreference.com/w/cpp/language/while
[18] https://en.cppreference.com/w/cpp/language/do
[19] https://en.cppreference.com/w/cpp/language/for

```
for (unsigned i = 0; i < 10; ++i) {
  // iterate 0, 1, 2, ..., 9
}
for (unsigned i = 0, len = getLength(); i != len; ++i) {
  // do something; doesn't call getLength() every iteration
}
for (unsigned i = 42; i-- > 0; ) {
  // iterate 41, 40, ..., 0
}
uint8_t i = 0;
for (; i < 256; ++i)
  std::println("{}", i); // hmmm....
```

Quiz: What could be a problem of the last loop?

A. No Problem B. Syntax Error C. Endless Loop D. Undefined Behavior

> Beware of integer overflows. Reminder: signed integer overflow is undefined behavior.

[Slide 50] Basic Functions[20]

- Associate a sequence of statements (body) with a name
- Function can have parameters and a return type (can be `void`)
- Non-`void` functions must execute `return` statement
- Arguments are passed **by value** (unlike Java for classes)
  - Pass-by-reference requires explicit annotation, see later

```
void procedure(unsigned parameter0, double parameter1) {
  // do something with parameter0 and parameter1
}
unsigned meaningOfLife() {
  // complex computation, takes 7.5 million years
  return 42;
}
```

[Slide 51] Basic Function Arguments

- Parameters can be unnamed ⇝ unusable, but still required on call
- Function can specify default arguments[21] in parameter list
  - After first param with default value, all must have a default value

```
unsigned meaningOfLife(unsigned /*unused*/) {
  return 42;
}
unsigned addNumbers(int a, int b = 2, int c = 3) {
  unsigned v = meaningOfLife(); // ERROR: expected argument
  unsigned w = meaningOfLife(123); // OK
  return a + b + c;
```

[20] https://en.cppreference.com/w/cpp/language/function
[21] https://en.cppreference.com/w/cpp/language/default_arguments

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

2.5. Namespaces

[Slide 52] Namespaces[22]

- Large projects contain many names ⇝ organize in logical units
- *namespaces* allow preventing name conflicts

```
namespace A {
void foo() { /* do something */ }
void bar() { foo(); /* refers to A::foo */ }
} // end namespace A
namespace B {
void foo() { /* do something */ }
} // end namespace B
int main() {
  A::foo(); // qualified name lookup
  B::foo(); // qualified name lookup
  foo(); // ERROR: foo was not declared in this scope
}
```

> Typically, the outermost namespace is used for the project name (e.g., `llvm`, `clang`, `umbra`). The namespace `std` is reserved for the C++ standard library. This prevents name collisions when using libraries.
>
> This has a big advantage over the C convention of using prefixes in names (e.g., `LLVM<name>` or `stdc_<name>`): inside namespaces, typing the redundant prefix can be avoided and namespaces can be imported with a `using namespace` directive (see below).

[Slide 53] Namespace Nesting

- Namespaces can be nested

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

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

[22] https://en.cppreference.com/w/cpp/language/namespace

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

[Slide 54] Namespaces: `using` and Conventions

- Typically: add comments to closing namespace brace
- Always using fully qualified names makes code easier to read
- But: sometimes, source is obvious and typing cumbersome...
  - `using namespace X;` imports *everything* from `X`
  - `using X::a;` imports only *a* from `X`

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

> Be careful about `using namespace`, this might pollute your namespace and result in unwanted naming collisions. You can also use `using` declarations inside a scope like this:
>
> ```
> namespace A { int x; }
> namespace B { int y; int z; }
> int main() {
>   using namespace A;
>   using B::y;
>
>   x = 1; // Refers to A::x
>   y = 2; // Refers to B::y
>   z = 3; // ERROR: z was not declared in this scope
>   B::z = 3; // OK
> }
> ```

2.6. Memory & Object Model

[Slide 55] Memory Model

- Fundamental storage unit: *byte*
  - There can (theoretically) be more than 8 bits in a byte
- Memory consists of one or more contiguous sequences of bytes
  - Memory can have holes, e.g. due to virtual memory
- Every byte has a unique address

[Slide 56] Objects[23]

- Object: region of storage; properties:
  - Size (see next slides)
  - Alignment (see next slides)
  - Storage duration (see next slides)
  - Lifetime (see next slides)
  - Type
  - Value
  - Optionally: name
- C++ programs create, destroy, refer to, access, and manipulate objects
- Examples for objects: local/global variables, parameters
  - Not objects: functions, references, values

[Slide 57] Object Size and Alignment

- Size and alignment requirements are defined by the type
- `sizeof` operator[24]: query size in bytes of object/type
  - `sizeof(char)` = `sizeof(std::byte)` = 1
  - All other sizes implementation-defined
- `alignof` operator[25]: query minimum alignment in bytes of type
  - Depending on implementation, some values must be aligned in memory
  - Alignment is always a power of 2
  - Address must be a multiple of the alignment

> Bytes that are larger than the industry standard of 8 bits are very rare, but do exist. Some embedded platforms, where the smallest possible memory access granularity is 32 bits, use 32-bit bytes. On such platforms, `sizeof(int)` can be 1.
>
> An alignment of x means that the address of the object is a multiple of x. The size is always a multiple of the alignment.

[Slide 58] Storage Duration[26]

- Every object has a storage duration

[23] `https://en.cppreference.com/w/cpp/language/object`
[24] `https://en.cppreference.com/w/cpp/language/sizeof`
[25] `https://en.cppreference.com/w/cpp/language/alignof`
[26] `https://en.cppreference.com/w/cpp/language/storage_duration`

Storage Duration	Begin	End	Note/Example
automatic	Scope begin	Scope end	Local variables
static	Program begin	Program end	Global variables
thread	Thread start	Thread end	`thread_local` vars
dynamic	`new`	`delete`	

- Static: allocated/initialized before `main` in non-guaranteed order[27]
- Thread: one copy of the object per thread
- Dynamic: allocation/deallocation must be done manually

[Slide 59] Lifetime[28]

Lifetime of an object...

- starts when it is fully *initialized*
- ends when destructor called (classes) or storage is deallocated/reused (others)
- never exceeds the lifetime of the storage (see storage duration)

- Using an object outside its lifetime is **undefined behavior**
- This is a main source of memory bugs
- Compilers can only warn about very basic errors

⇒ If compiler warns, always **fix your program**

> When the compiler warns about a possible lifetime bug, this is most likely a problem in your code. Again: fix it. There can be very rare occasions where the warning is a false positive, but in these cases, you should adjust your code nonetheless so that the warning disappears.
>
> The lifetime of a reference (see later) ends as if it were a scalar object (e.g., `int`).

[Slide 60] Lifetime: Example

Quiz: When does the lifetime of `p` end?

```
int g;
void matterOfLifeOrDeath(unsigned a) {
  thread_local int t = 1;
  unsigned c = a;
  {
    unsigned p = a + 1;
  }
  unsigned m = t - 1;
}
```

A. At the end of the function.

[27] https://en.cppreference.com/w/cpp/language/siof
[28] https://en.cppreference.com/w/cpp/language/lifetime

B. At the end of the innermost block.
C. At the end of the program.
D. When the underlying stack space is reuseed (e.g., for `m`).

[Slide 61] Lifetime: Example

Quiz: What is problematic about this function?

```
int fancyZero() { // fancy way to return zero
  int x = x ^ x;
  return x;
}
```

A. Ill-formed/compile error: `x` used before its declaration.
B. Undefined behavior: signed integer overflow.
C. Undefined behavior: `x` used outside its lifetime.
D. Undefined behavior: `x` used outside its storage duration.

[Slide 62] Basic Syntax and Object Model – Summary

- Fundamental types: void, integral, floating-point
- Exact width, representation, etc. not specified by standard
- Undefined behavior means anything can happen
- Undefined behavior must therefore never happen
- Basic syntax similar to other C-like languages, with additions
- Use namespaces to avoid naming collisions
- C++ programs resolve around working with objects
- Objects' lifetime is often implicit, leading to subtle bugs

[Slide 63] Basic Syntax and Object Model – Questions

- What is the required minimum size of an `unsigned int`?
- Why is arithmetic on `char` problematic?
- Why is `long double` rarely used?
- What can happen when undefined behavior is encountered?
- How can compilers use undefined behavior for optimizations?
- Which variable initializer prevents loss of accuracy?
- What is the storage duration of an object?
- What is the relation between storage duration and lifetime?

3. Declarations/Definitions, Preprocessor, Linker

[Slide 65] On "Internet"

Search engines/AI are **not** your friend when it comes to C++!
Use high-quality sources. Use the C++ reference. Read the script of this lecture.

3.1. Preprocessor

[Slide 66] Compiler: Overview (1)


```
clang++ -E -o hello.ii hello.cpp
clang++ -o hello hello.ii
```

- Preprocessor transforms source code before actual compilation
- `clang++ -E` – stop after preprocessing

[Slide 67] Preprocessor[1]

- Applies textual transformation before compilation
  - E.g., to conditionally exclude certain code paths from compilation
  - Preprocessor has no knowledge about "real" C++ language semantics
- Handles preprocessor directives: lines that begin with `#`
- Outputs program without directives

Use **carefully**, avoid if possible!

[Slide 68] Preprocessor: `#include`[2]

- `#include "path/to/file"` – copy content from file at current position
- Literal textual inclusion ("copy-paste")

[1] `https://en.cppreference.com/w/cpp/preprocessor`
[2] `https://en.cppreference.com/w/cpp/preprocessor/include`

```
//--- magicNumber.inc
42

//--- magicNumber.cpp
int magic =
#include "magicNumber.inc"
  ;
```

- After preprocessing

```
// clang++ -E magicNumber.cpp
int magic =
42
  ;
```

[Slide 69] Preprocessor: Include Path

- `#include "file"`
  - Search order: current directory, include path, system path
  - Convention: use for files in current project
- `#include <file>` – search include path, then system path
  - Search order: include path, system path
  - Convention: use for libraries and system includes
- Compiler flag: `-I<directory>` – add directory to include path
- CMake: `target_include_directories(target PUBLIC src/)`
- Typical: add root of project source to include path

  ⇒ All files can be included by "absolute path"

[Slide 70] Preprocessor: `#define`[3]

- `#define SOMENAME` – define a macro with the given name
- Can have an optional textual replacement
- `#undef` – undefined previously defined macro

```
#define EMPTY
#define return never
#define ANSWER 42
#define FUNC_DECL int getAnswer()
#undef return
FUNC_DECL { EMPTY return ANSWER; EMPTY }
// Preprocessed to:
// int getAnswer() { return 42; }
```

Don't use the preprocessor like this, this is primarily for illustration.[4]

[Slide 71] Preprocessor: `#define` – Example

[3] `https://en.cppreference.com/w/cpp/preprocessor/replace`

[4] NB: Re-defining keywords is undefined behavior if the standard library is included.

Quiz: What does the function `f` return?

```
#define ONE 1
#define TWO (ONE + ONE)
#define FOUR TWO+TWO
#define SIXTEEN FOUR*FOUR
int f() { return (SIXTEEN + FOUR) * TWO + TWO; }
```

A. (compile error) B. 2 C. 26 D. 42

Don't use the preprocessor like this, this is primarily for illustration.

> When placing expressions of any kind in a macro, it is highly recommendable to wrap them with parenthesis.

[Slide 72] Preprocessor: Pre-defined Macros

- Some macros are pre-defined by the compiler
- Few are standardized, others vary between compilers
- Typically begin with double-underscore

Examples:

- `__cplusplus` – used C++ standard, e.g. `202302L`
- `__FILE__` – name of the current file
- `__x86_64__` – defined if compiling for x86-64
- Compiler flag `-D<macroname>=<expansion>` – define a macro with the (optional) expansion

> Typically, many macros are pre-defined to describe the environment. You can use `clang++ -dM -E -x c++ - </dev/null` to list all pre-defined macros and their expansions. (You can use `clang++ -- help` to understand the supplied command line flags.)

[Slide 73] Preprocessor: Conditions[5] (1)

- `#if <expr>`/`#elif <expr>`/`#ifdef <macro>`/`#ifndef <macro>`/`#else`/`#endif` – conditionally compile part of code
  - Use cases: architecture-dependent code, code only for debug builds
- Expressions can use `defined(MACRO)` to test whether a macro is defined
- Preprocessor expressions *only* operate on macros!

```
#if defined(__x86_64__)
// x86-64-specific code goes here
#elif defined(__aarch64__)
// aarch64-specific code goes here
#else
// architecture-independent code goes here
#endif
```

[5] https://en.cppreference.com/w/cpp/preprocessor/conditional

[Slide 74] Preprocessor: Conditions (2)

- `#error` – cause compilation to fail with given message

```
#if defined(__x86_64__) || defined(__aarch64__)
// x86-64 and aarch64 code goes here
#else
#error Unsupported architecture!
#endif

#if 0 // #if 0 can be used for comments, can be nested (unlike /* */)
void commentedOut() {}
#if 0
void moreCommentedCode() {}
#endif
#endif
```

[Slide 75] Preprocessor: Conditions (3)

Quiz: What does the function `f` return?

```
int j = 5;
#if j * j == 25
int f() { return j * j; }
#else
int f() { return 20; }
#endif
```

A. (compile error) B. depends on `j` C. 20 D. 25

Don't use the preprocessor like this, this is primarily for illustration.

[Slide 76] Preprocessor: Function-Like Macros

- Macros can have arguments, so they look like functions
- Again, purely textual replacement, no semantics
  - Wrap all parameters in parentheses to avoid precedence issues

```
#define MIN(a,b) ((a)<(b)?(a):(b))

int min3(int a, int b, int c) {
// Preprocessed to:
// return ((((a)<(b)?(a):(b)))<(c)?(((a)<(b)?(a):(b))):(c));
  return MIN(MIN(a, b), c);
}
```

Don't use the preprocessor like this, this is primarily for illustration.

[Slide 77] Preprocessor: Function-Like Macros (Quiz)

Quiz: Why is this macro problematic?

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))
```

A. One parameter is evaluated multiple times.
B. The unnecessary parenthesis make the code difficult to read.
C. The macro doesn't compute the minimum on unsigned integers.

- Don't do this — we'll cover modern replacements later

[Slide 78] Preprocessor: Recommendations

Avoid if possible!

- Many pitfalls, code harder to read/analyze/debug
- Many use-cases have modern, safer C++ replacements (see later)

- No rule without exceptions...
- Some older code bases use preprocessor heavily
  - Primary reason we cover it so extensively here

> - Use `constexpr` global variables instead of `#define BAR 1`
> - Use type-generic functions for function-like macros
> - Use `if constexpr ()` instead of `#if`/`#endif`
>
> There are, of course, exceptions to these guidelines. But generally, avoid the use of the preprocessor and only use it for header guards and header includes.

3.2. Assertions

[Slide 79] Runtime Checks for Debugging: `assert`

- `assert(expr)` – abort program if assertion is false
- Use to check invariants
- When `NDEBUG` is defined, assert generates no code
- CMake automatically defines `NDEBUG` in release builds

```
#include <cassert>
double div(double a, int b) {
  assert(b != 0 && "divisor must be non-zero");
  return a / b;
}
```

> The idiom `assert(condition && "explanation")` is widely used to add a more helpful message or reasoning to the assertion. This works, because `"strings"` always get converted to a non-zero value (simplified, will be explained later together with pointers).

[Slide 80] `assert` – Implementation

- `assert(expr)` is a preprocessor macro

⇒ Expression gets *removed from source code* when `NDEBUG` is defined!

```
//--- /usr/include/assert.h (glibc) (excerpt) (code simplified for slide)

/* void assert (int expression);

   If NDEBUG is defined, do nothing.
   If not, and EXPRESSION is zero, print an error message and abort. */
#ifdef NDEBUG
# define assert(expr) ((void)(0))
#else
# define assert(expr) ((expr) ? (void)(0) : __assert_fail(#expr, /*...*/))
#endif
```

> `#expr` is stringifies the parameter `expr`, which is the string printed to the console when the assertion fails.

3.3. Declaration & Definitions

[Slide 81] Multiple Source Files

- C++ source files know nothing about each other
  - Other than `#include`, which is just copy-paste

How do they know what functions other files define?

⇝ Explicit declarations

[Slide 82] Declarations[6]

- Declarations introduce names
- Names must be declared before they can be referenced
- Variables: `int x;`
- Functions: `void fn();`
- Namespace: `namespace A { }`
- Using: `using A::x;`
- Class: `class C;`
- ...

[Slide 83] Definition[7]

- A declared name can be used, but: most uses require[8] a *definition*
  - Reading/writing value or taking address of an object
  - Calling or taking address of function
- Most declarations are also definitions, with some exceptions

[6] `https://en.cppreference.com/w/cpp/language/declarations`
[7] `https://en.cppreference.com/w/cpp/language/definition`
[8] Formally called *odr-use*

  - Function declaration without body
  - Variable declarations with `extern` and no initializer

[Slide 84] Function Declarations: Example

- Forward declaration necessary for cyclic dependencies

```
void bar(int n); // declaration, no definition

void foo(int n) { // declare + define foo
  std::println("foo");
  if (n > 0)
    bar(n - 1); // OK, bar declared above
}

void bar(int n) { // re-declare + define bar
  std::println("bar");
  if (n > 0)
    foo(n - 1); // OK, foo declared above
}
```

> Without the forward declaration of `bar`, compilation will fail, because `bar` is not declared at the function call inside `foo`.
>
> It is generally advisable to avoid cyclic dependencies.

[Slide 85] Variable Declarations: Example

```
extern int global; // declaration
int otherGlobal; // declarartion + definition, zero-initialized

int readGlobal() {
  return global;
}

int global = 5; // re-declaration + definition
```

- The first declaration is rather useless, could move definition there

[Slide 86] cv-Qualifier: `const` and `volatile`[9]

- Part of the type, can appear in variable declarations
- `const` – object cannot be modified
- `volatile` – object access has a side-effect
  - E.g., direct hardware access, communication with signal handlers

```
void function() {
  int a = 4;
  const int b = a;
  a = 0; // OK
  b = 10; // ERROR: assignment of read-only variable
  volatile int v = 5; // will not be optimized out
```

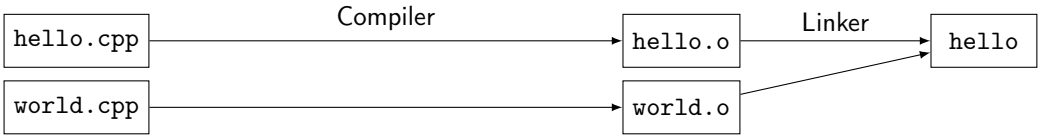
[9] `https://en.cppreference.com/w/cpp/language/cv`

```
}
```

> Do not use `volatile` unless you know that it is strictly require. *Do not use* volatile for synchronization across multiple threads!

3.4. Linker

[Slide 87] Compiler: Overview (2) – Multiple Files




```
clang++ -c -o hello.o hello.cpp
clang++ -c -o world.o world.cpp
clang++ -o hello hello.o world.o
```

- Compiler generates object file with machine code
  - One compile invocation compiles one *translation unit*
  - May contain references to not-yet-defined functions/globals
- Linker combines object files into executable
  - Resolve all undefined references

> The compiler invocation `clang++ -o hello hello.cpp world.cpp` internally runs all steps, but discards the intermediate results. This is impractical, because if `hello.cpp` changes but `world.cpp` did not, both need be recompiled.
>
> Separating the compiler invocations (`clang++ -c -o hello.o hello.cpp`; `clang++ -c -o world.o world.cpp`; `clang++ -o hello hello.o world.o`) allows selectively rebuilding the parts of the program that changed, significantly reducing the times of incremental builds. The compiler option `-c` instructs the compiler to stop after the compilation and emit an object file (instead of linking an executable).

[Slide 88] Multiple Files

```cpp
//--- foo.cpp
int globalVar = 7;
int foo() { return 6; }

//--- main.cpp
#include <print>
extern int globalVar;
int foo();
int main() {
  std::println("{}", globalVar * foo());
  return 0;
}
```

```
$ clang++ -std=c++23 -c -o foo.o foo.cpp
$ clang++ -std=c++23 -c -o main.o main.cpp
$ clang++ -o main main.o foo.o
$ ./main
42
```

> Declaration and definitions can be in different files. The compiler needs a declaration to know that a function/variable will exist, but does not require a definition. The definition must be supplied at link time.

[Slide 89] Multiple Files: Undefined References

```
//--- foo.cpp
int bar();
int foo() { return 2 * bar(); }

//--- main.cpp
extern int undefinedGlobal;
int main() {
  return undefinedGlobal;
}
$ clang++ -std=c++23 -c -o foo.o foo.cpp
$ clang++ -std=c++23 -c -o main.o main.cpp
$ clang++ -o main main.o foo.o
/usr/bin/ld: main.o: in function ‘main’:
main.cpp:(.text+0x8): undefined reference to ‘undefinedGlobal’
/usr/bin/ld: foo.o: in function ‘foo()’:
foo.cpp:(.text+0x8): undefined reference to ‘bar()’
clang++: error: linker command failed with exit code 1 (use -v to see invocation)
```

> If a global variable or function is referenced, but not defined in any of the object files (or libraries, including the standard library), the linker will detect this and fail. The compiler cannot detect this, as it has no knowledge about other object files or used libraries.

3.5. One Definition Rule

[Slide 90] One Definition Rule (ODR)[10]

- At most one definition of a name allowed *within one translation unit*
- Exactly one definition of every used function or variable must appear *within the entire program*
- (for more cases, exceptions, subtleties: see reference documentation)

NB: Some ODR violations make programs “ill-formed, no diagnostic required” — only the linker can diagnose such violations

[10] https://en.cppreference.com/w/cpp/language/definition#One_Definition_Rule

[Slide 91] One Definition Rule: Examples (Multiple Definitions)

```
int i = 0; // OK: declaration + definition
int i = 1; // ERROR: redefinition
```

```
//--- a.cpp
int foo() { return 1; }

//--- b.cpp
int foo() { return 2; }
clang++ -std=c++23 -c -o a.o a.cpp
clang++ -std=c++23 -c -o b.o b.cpp
clang++ a.o b.o
/usr/bin/ld: foo.o: in function `foo()':
b.cpp:(.text+0x0): multiple definition of `foo()'; a.o:a.cpp:(.text+0x0): first defined
    here
```

3.6. Header and Implementation Files

[Slide 92] Header and Implementation Files

- Duplicating declarations into every file technically possible
- But: not maintainable, error-prone

Idea: split into *implementation* (`.cpp`) and *header* (`.h`) file:

- Header file: just declarations that should be usable in other files
  - Conceptually: "API" of logical unit
  - Also should include documentation
- Implementation file: definitions for names declared in header
  - Conceptually: "implementation" of the API

Use *preprocessor* to copy-paste declaration

[Slide 93] Header and Implementation Files: Example

```
//--- sayhello.h
#include <cstdint>
/// Print "Hello!" to standard output.
void sayHello(std::uint64_t number);

//--- sayhello.cpp
#include "sayhello.h"
#include <cstdint>
#include <print>
void sayHello(std::uint64_t number) { std::println("Hello␣{}!", number); }

//--- main.cpp
#include "sayhello.h"
int main() { sayHello(1); return 0; }
```

[Slide 94] Header Guards

- Header files include other headers they require

  - E.g., for defined data types (see later)
- Transitive includes: same header might be included multiple times!
- But: can cause problems due to redefinitions

⇝ Wrap entire header with `#ifdef` and unique identifier

```
//--- sayhello.h
#ifndef CPPLECTURE_HELLO_H
#define CPPLECTURE_HELLO_H

/// Print "Hello!" to standard output.
void sayHello();

#endif // CPPLECTURE_HELLO_H
```

- Non-standard alternative

```
//--- sayhello.h
#pragma once

/// Print "Hello!" to standard output.
void sayHello();
```

> The first time `sayhello.h` is included, the macro `CPPLECTURE_HELLO_H` is not defined and therefore the remainder of the file will be considered. In particular, the macro will be defined. In a possible later second inclusion, the macro will be defined and the content of the file will be ignored.
>
> It is crucial that every header has a unique name for the header guard macro. By convention, the name of the path and file is used. This is particularly important when copying files.
>
> `#pragma once` is an alternative for the same goal. However, as there is no portable way to actually determine whether two files are the same (consider symbolic links, hard links, etc.), it is not standardized and therefore avoided by many projects.

[Slide 95] Header Files and `#include`

- Include (exactly) used header files at the beginning
  - In both, header and implementation file
  - Be careful about transitive includes
- Typically grouped by: (Example)
  1. Accompanying header file
  2. Project includes
  3. Library includes
  4. System includes
- Only include header files
- **Never** include implementation files!

[Slide 96] Typical Project Layout

```
+-- CMakeLists.txt
+-- src/
    +-- Module.cpp
    +-- Module.hpp
    +-- Printer.cpp
    +-- Printer.hpp
    +-- log/
        +-- Log.cpp
        +-- Log.hpp
        +-- LogEntry.cpp
        +-- LogEntry.hpp
    +-- main.cpp
```

- Source files and header files next to each other
- Entry points (`main()`) often separate
  - Typically small files ⇝ easier testing
- CMakeLists defines
  - `add_executable` with all sources (`*.cpp`)
  - `target_include_directories(... src)`
- Alternative layouts exist

> Some typical variants from this layout:
>
> - Headers are stored in a separate `include` directory next to `src`.
>   This is typically seen with libraries, where the public headers, which get installed and exposed to library users, are in `include` while private headers are in `src`.
> - Programs with `main` reside in a separate directory (e.g., `tools`). The main source files are compiled as a static library and executables link against the static library.
>   This is typically used when multiple programs get compiled. This way, the main source files get compiled only once and are testable, because they don't provide a program entry point.
> - One `CMakeLists.txt` per directory, which adds the source files of the directory to some variable instead of having a single top-level file that lists all files.
>   This is typically used by large projects where a single file list might not be maintainable.

[Slide 97] Tracking Changes in Source Code

```cpp
//--- a.hpp
extern int globalA;
//--- a.cpp
#include "a.hpp"
int globalA = 10;
//--- square.hpp
#include "a.hpp"
int square(int n = globalA);
//--- square.cpp
#include "square.hpp"
void square(int n) {
```

```
  return n * n;
}
//--- main.cpp
#include "square.hpp"
// ...
```

Quiz: `a.hpp` changed. Which files to re-compile?

A. `a.hpp`
B. `a.cpp`
C. `a.cpp`, `square.cpp`
D. `a.cpp`, `square.cpp`, `main.cpp`
E. `a.hpp`, `a.cpp`, `square.cpp`, `main.cpp`

[Slide 98] Tracking Changes in Source Code

- Incremental compilation: only recompile files that actually changed
  - Substantially reduces build time during development
- Detecting files that need recompilation is non-trivial
  - Transitive dependencies of header files
- Build systems like CMake use compiler to output list of used includes
  - If any of the files changed, the source file needs recompilation

> It is also possible to achieve accurate tracking of updated header files with plain `Makefile`s, but it is non-trivial (it involved instructing the compiler to write dependencies to separate files and including these files in the `Makefile`). Thus, it is strongly recommendable to use a proper build system for C++ projects, which track dependencies. Examples are CMake, Meson, scons, and waf, and several others exist as well.

3.7. Linkage

[Slide 99] Linkage

- Linkage of declaration: visibility across different translation units
- No linkage: name only usable in their scope
  - E.g., local variables
- Internal linkage: can only be referenced from same translation unit
  - Global functions/variables with `static`
  - const-qualified global variables without `extern`
  - Declarations in namespace without name (“anonymous namespace”)
- External linkage: can be referenced from other translation units
  - Global functions/variables (unless `static`)

[Slide 100] Declaration Specifiers

- Variable/function declarations allow for additional specifier
- Controls storage duration *and* linkage

Specifier	Global Func/Variable	Local Variable
none	static + external	automatic + none
`static`	static + internal	static + none
`extern`	static + external	static + external
`thread_local`	thread + ext/int	thread + none

- And there's `inline` (it deserves it's own slide)

[Slide 101] Declaration Specifiers – Example

```
//--- a.cpp
static int foo = 1;
int bar = 2;
static int add(int x, int y) { return x + y; }
int countMe() {
  static int counter = 0; // static storage duration, no linkage
  return counter++;
}

//--- b.cpp
static int foo = 1; // OK
int bar; // ERROR: ODR violation

// OK: a.cpp's and b.cpp's add have internal linkage
static int add(int x, int y) { return x + y; }
```

> You can use `static` on local variables. This can be useful for constant data that should only be visible inside the current function. Mutable variables with static storage duration are problematic in practice when multiple threads call the function in parallel.
>
> As a general advice, avoid mutable global variables (or local `static` variables) for this reason.

[Slide 102] Internal Linkage: Anonymous Namespaces

- Option A: Use `static` (only works for variables and functions)
  ```
  static int foo = 1; // internal linkage
  static int bar() { // internal linkage
      return foo;
  }
  ```
- Option B: Use *anonymous namespaces* (preferred)
  ```
  namespace {
  int foo = 1; // internal linkage
  int bar() { // internal linkage
  ```

```
    return foo;
}
} // end anonymous namespace
```

> In C++, prefer anonymous namespaces over `static` to change the linkage of global declarations.

[Slide 103] `inline` Specifier[11]

- `inline` – permit multiple definitions in different translation units
  - No direct relation to the inlining optimization!

```
//--- sum.h
#ifndef SUM_H
#define SUM_H

inline int sum(int a, int b) {
  return a + b;
}

#endif // SUM_H
//--- a.cpp
#include "sum.h"
// Now has definition of sum
// ...

//--- b.cpp
#include "sum.h"
// Now has definition of sum
// ...
```

- Linker keeps only one definition

> Inline definitions are useful when the presence of the definition can improve optimization, e.g. give the compiler the *opportunity* to do inlining, which would be impossible if the definition was in a different translation unit.
>
> As a downside, the function gets compiled in every translation unit that includes the definition, increasing compilation times and the size of the intermediate object files.
>
> Especially for inline definitions it is important to use header guards to prevent multiple definitions in the *same* translation unit.

[Slide 104] Declarations/Definitions, Preprocessor, Linker – Summary

- Preprocessor transforms source code before actual compilation
  - Use (almost) exclusively for header guards and header includes
- Use `assert()` for invariants, but be aware that it is a macro

[11] `https://en.cppreference.com/w/cpp/language/inline`

- Declarations introduce names, but not necessarily define them
- Exactly one definition of used func/var required in program
- For multiple files, separate header and implementation files
- There must be exactly one definition of every used name
- Exceptions: internal linkage and inline functions

[Slide 105] Declarations/Definitions, Preprocessor, Linker – Questions

- Why is the use of function-like macros problematic?
- What are state modifications inside `assert()` problematic?
- What is the difference between a declaration and a definition?
- How to declare functions and global variables?
- Why is the header guard important?
- Why is including C++ implementation files (`.cpp`) a bad idea?
- What does the `static` specifier do on local variables?
- What is the effect of an unnamed namespace?

4. References, Arrays, Pointers

[Slide 107] Value Categories (Simplified)

lvalue

- Can appear on *left* side of assignment
- Locates an object
- Has an address
- Examples:
  - Variable names: `var`
  - Assignment exprs: `a = b`

rvalue

- Can only appear on *right* side of assignment
- Might not have address
- lvalue can be converted implicitly to rvalue
- Examples:
  - Literals: `42`
  - Most exprs: `a + b`, `a < b`

> This is a simplified version of the value categories, mostly coherent with very old C++ standards prior to C++11. We will cover the all possible types of values when discussing move semantics in a later lecture.
>
> Not all lvalues can actually be assigned to, for example, `const`-qualified variables cannot be modified.

4.1. References

[Slide 108] Reference Declarations (1)[1]

- Declare an alias to an existing object or function
- Lvalue reference: `type& declarator`
- Definitions must be initialized to refer to a valid object/function
- Declarations don't need initializer, e.g. parameters
- Peculiarities:
  - References are immutable, i.e. can't change which object is aliased
  - References are not objects

  ⇒ No references to references

[Slide 109] Lvalue References: Example (Alias)

```
unsigned i = 10;
unsigned j = 20;
unsigned& r = i; // r is now an alias for i
```

[1] `https://en.cppreference.com/w/cpp/language/reference`

```
r = 15; // modifies i to 15
r = j; // modifies i to 20

i = 42;
j = r; // modifies j to 42
```

[Slide 110] Lvalue References: Example (Pass By Reference)

- References are used to implement pass-by-reference semantics

```
#include <print>
void computeAnswer(int& result) {
  result = 42;
}

int main() {
  int theAnswer = -1;
  computeAnswer(theAnswer); // theAnswer is now 42
}
```

[Slide 111] Lvalue References: Example (Returning Reference)

- Function calls returning lvalue references are lvalues

```
int global1 = 0;
int global2 = 0;

int& getGlobal(int num) {
  if (num == 1)
    return global1;
  return global2;
}

int main() {
  getGlobal(1) = 10; // global1 is now 10
  getGlobal(2)--; // global2 is now -1
}
```

> A typical application of this is not to return references to globals but to return references to class members.

[Slide 112] References and cv-Qualifiers

- References themselves cannot be cv-qualified
- But the referenced type can be
  - Reference can be initialized by less cv-qualified type e.g. `const int&` can be initialized from `int&`

```
#include <print>

void printAnswer(const int& answer) {
  std::println("{}", answer);
}
```

```
int main() {
  int theAnswer = 42;
  printAnswer(theAnswer); // cannot modify theAnswer
}
```

> For primitive types, passing by constant reference is rather pointless. It is important, however, when passing more complex objects, which could be non-trivial or expensive to copy.

[Slide 113] Pass-By-Reference

Quiz: What is the output of the program?

```
#include <print>
void foo(const int& a, int& b, const int& c) {
  b += a;
  b += c;
}

int main() {
  int x = 1;
  foo(x, x, x);
  std::println("{}", x);
}
```

A. (undefined behavior) B. 1 C. 2 D. 3 E. 4

[Slide 114] Dangling References[2]

- Lifetime of object can end while references still exist

$\rightsquigarrow$ dangling reference, when used: undefined behavior

```
int& foo() {
  int i = 123;
  return i; // DANGER: returns dangling reference
}
int bar() {
  int& res = foo();
  return res; // object used outside its lifetime => UB
}
```

[Slide 115] Rvalue References

- Extend the lifetime of temporary objects
  - NB: const lvalue references can also extend lifetime of temporaries
- Rvalue reference: `type&& declarator`
- Cannot bind directly to lvalues

[2] https://en.cppreference.com/w/cpp/language/reference#Dangling_references

```
int i = 10;
int&& j = i; // ERROR: cannot bind lvalue
int&& r = 42; // OK

int&& k = i + i; // OK, k == 20
k += 22; // OK, k == 42

const int& l = i * i; // OK, l == 100
l += 10; // ERROR: cannot modify constant reference
```

> Typically, the lifetime of temporary objects (if they exist), ends at the end of the full expression. Rvalue references extend the lifetime of such temporaries until they go out of scope.

[Slide 116] Passing Rvalues

Quiz: What is the output of the program?

```
#include <print>
int foo(const int& a, const int& b, int&& c) {
  c += b;
  return c + a;
}

int main() {
  int x = 1;
  int r = foo(x, x, x);
  std::println("{}", r);
}
```

A. (compile error) B. 1 C. 2 D. 3 E. 4

[Slide 117] Passing Rvalues

Quiz: What is the output of the program?

```
#include <print>
int foo(const int& a, const int& b, int&& c) {
  c += b;
  return c + a;
}

int main() {
  int x = 1;
  int r = foo(x, x * 2, x + 10);
  std::println("{}", r);
}
```

A. (compile error) B. (undefined behavior) C. 13 D. 14 E. 26

[Slide 118] Reference Declaration Syntax

- `&` and `&&` syntactically belong to the declarator!

```
int i = 10;
int& a = i, k = 2; // a is int&, k is int
```

⇒ Only declare one identifier at a time!

- `int& j = 1;` and `int &j = 1;` are valid, follow code style

[Slide 119] Rvalue References: Overload Resolution

```
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&&)
}
```

> We did not introduce overloads so far in this lecture. A function of one name can have multiple overloads with different parameter counts types. Overload resolution is the process of determining which function gets called. As this also considers implicit conversions, it is a rather complex procedure, which we will cover later (at least to some extent).
>
> It is typically not advisable to implement different semantics for const vs. non-const references. Overloads that take rvalue references will later be used for move semantics (as the parameter must be an rvalue, it is "lost" after the function call anyway).

4.2. Arrays

[Slide 120] Arrays[3]

- Syntax (C-style arrays): `type declarator[expression];`
- *expression* must be an integer constant **at compile-time**
- Elements can be accessed with `[]` with index $0 \cdots < N$
- Arrays cannot be assigned or returned

```
unsigned short arr[10];
for (unsigned i = 0; i < 10; ++i)
  arr[i] = i * i;
```

[3] `https://en.cppreference.com/w/cpp/language/array`

```
unsigned a[10];
unsigned b[10];
a = b; // ERROR: cannot assign arrays
```

> As for references, the array-size is part of the declarator and doesn't belong to the type specifier.

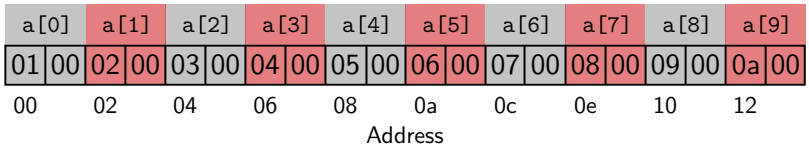
[Slide 121] Array Initialization

- Without an initializer, elements are default-initialized
  - Remember: for local variables, this means uninitialized
- Zero-initializer:
  ```
  unsigned short arr[10] = {}; // 10 zeroes
  ```
- List-initializer:
  ```
  unsigned short arr[] = {1, 2, 3, 4, 5, 6}; // 6 elements
  ```

[Slide 122] Array Memory Layout

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

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



Arrays are just dumb chunks of memory

- Out-of-bounds accesses are not detected
- May lead to rather weird bugs, not necessarily crashes
- Exist mainly due to compatibility requirements with C

[Slide 123] `sizeof` Array

- Like for other types: `sizeof` return array size **in bytes**
- Divide by size of an element to determine array length

```
unsigned short a[10];

for (unsigned i = 0; i < sizeof(a) / sizeof(a[0]); ++i)
  a[i] = i * i;
```

(Don't do this in C++)

> This way of determining the size of an array is inherited from C, but inherently error-prone. In C++, don't do this. Instead, use `std::array` (see later), which

> provides a `size()` method.

[Slide 124] Multi-Dimensional Arrays

- Array elements can be arrays themselves

```
unsigned md[3][2]; // array with 3 elements of (array of 2 unsigned int)
for (unsigned i = 0; i < 3; ++i)
  for (unsigned j = 0; j < 2; ++j)
    md[i][j] = 3 * i + j;

unsigned b[][2] = { // only the outermost dimension can be omitted
  {0, 1},
  {2, 3},
  {4, 5},
};
```

- Elements still allocated contiguously in memory

> For multi-dimensional arrays, the size of the innermost dimension comes *last*.

[Slide 125] `size_t`[4]

- Designated types for indexed and sizes: `std::size_t` (`<cstddef>`)
- Unsigned integer type large enough to represent all possible array sizes and indices on the target architecture
- Used throughout the standard library for indices/sizes
- Generally use `size_t` for indexes and array sizes
  - For small arrays, `unsigned` might be sufficient
  - Do not use `int`

[Slide 126] `std::array`[5]

C-style arrays should be avoided whenever possible

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

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

[4] https://en.cppreference.com/w/cpp/types/size_t
[5] https://en.cppreference.com/w/cpp/container/array

[Slide 127] `std::array`

- ... can be returned (unlike C-style arrays)

```
std::array<int, 10> squares() {
  std::array<int, 10> res = {}; // zero-initialize all elements
  for (size_t i = 0; i < a.size(); ++i)
    res[i] = i * i;
  return res;
}
```

- ... can be passed as parameter (unlike C-style arrays)

```
// NB: src is copied by value, might be expensive!
// Prefer const std::array<int, 10>& src instead. (btw, don't write this code)
void copy(std::array<int, 10>& dst, std::array<int, 10> src) {
  assert(dst.size() == src.size() && "size␣mismatch!");
  for (size_t i = 0; i < dst.size(); ++i)
    dst[i] = src[i];
}
```

> Be very careful about passing arrays by value! Generally, don't do this unless there's a good reason. Small arrays (e.g., two integers) are typically fine, larger arrays can incur a substantial performance penalty.
>
> Don't write a `copy` function like this; instead, rely on standard library function (e.g., `std::memcpy`, `std::copy`), which tend to be more optimized.

[Slide 128] For-Range Loop

- Syntax: `for (range-declaration : range-expression) loop-statement`
- Execute loop body for every element in range expression

```
std::array<int, 3> a = {1, 2, 3};
for (int& elem : a)
  elem *= 2;
// a is now {2, 4, 6}

for (const int& elem : a)
  std::println("{}", elem);
```

[Slide 129] Special Case: String Literals

- String literals are immutable null-terminated character arrays
- Type of literal with N characters is `const char[N+1]`
- Artifact of C compatibility
- Generally avoid, use `std::string_view` or `std::string` instead
- Occasionally needed for interfacing with C APIs

[Slide 130] String Literals

Quiz: What does the function `f` return?

```
size_t f() { return sizeof("Hello!"); }
```

A. (compile error) B. impl.-defined C. 5 D. 6 E. 7

4.3. Pointers

[Slide 131] Pointers[6]

- Syntax: `type* cv declarator`
  - As for references/arrays/functions, the `*` is part of the declarator
- No pointers to references, cv qualifies the pointer itself
- Points to an object, stores *address* of first object byte in memory
- Pointers are objects (unlike references)
- Like reference, pointers can dangle

```
int* a; // pointer to (mutable) int
const int* a; // pointer to const int
int* const a; // const pointer to (mutable) int
const int* const a; // const pointer to const int

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

[Slide 132] Address-Of Operator[7]

- Operator `&`: obtain pointer to object
- Opeand must be an lvalue expression, cv-qualification is retained

```
int a = 10;
int* ap = &a;
const int c = 20;
const int* cp = &c;
int* cp2 = &c; // ERROR: cannot convert const int* to int*

int& r = a; // Reference to a
int* rp = &r; // Pointer to a
```

[Slide 133] Indirection Operator[8]

- Operator `*`: obtain lvalue reference to pointed-to object
- Operand must be a pointer, cv-qualification is retained
- Also referred to as *pointer dereference*

```
int a = 10;
int* ap = &a;
```

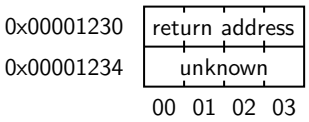
[6] https://en.cppreference.com/w/cpp/language/pointer
[7] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_address-of_operator
[8] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_indirection_operator

```
int& ar = *ap;
ar = 20; // a is now 20
*ap = 4; // a is now 4
```

[Slide 134] What is Happening? (1)

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

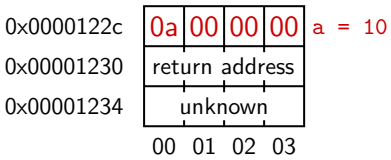
Stack Memory



[Slide 135] What is Happening? (2)

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

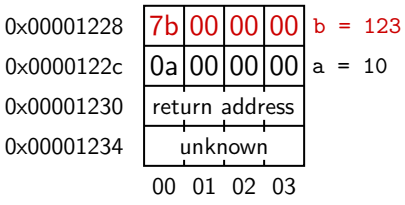
Stack Memory



[Slide 136] What is Happening? (3)

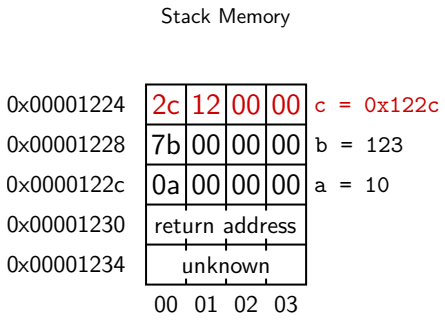
```
int main() {
    int a = 10;
    int b = 123;
}
```

Stack Memory



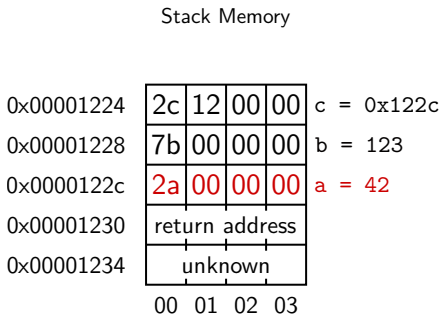
[Slide 137] What is Happening? (4)

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



}
```



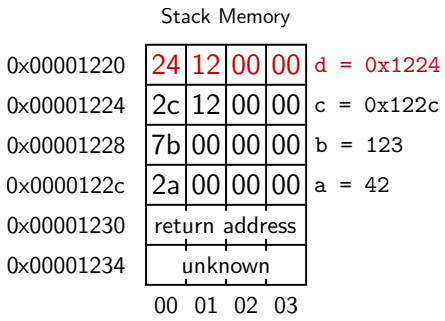
[Slide 138] What is Happening? (5)

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



}
```



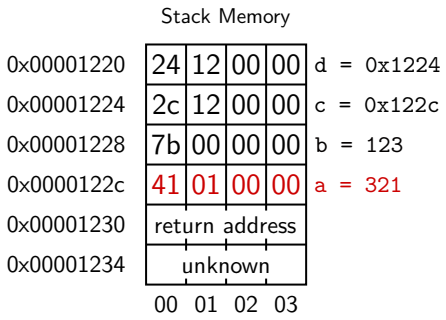
[Slide 139] What is Happening? (6)

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



}
```



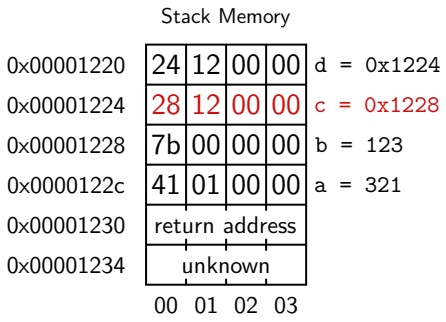
[Slide 140] What is Happening? (7)

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



}
```



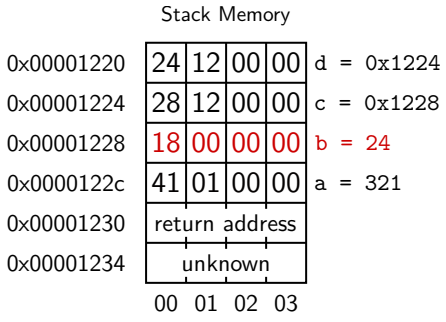
[Slide 141] What is Happening? (8)

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



}
```



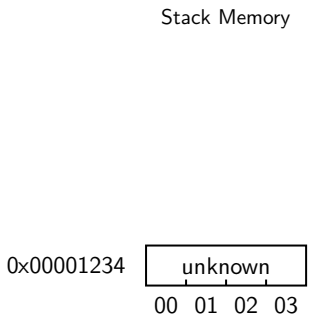
[Slide 142] What is Happening? (9)

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



}
```



[Slide 143] What is Happening? (10)

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

    return 0;
}
```



[Slide 144] Pointers to References?

Quiz: Why are pointers to references impossible?

A. References are not objects and thus have no address.
B. Would be redundant to pointers to pointers.
C. Taking the address of the referenced object is not possible.

[Slide 145] Null Pointers[9]

- Pointer can point to object, or nowhere (null pointer)
- Null pointer has special value `nullptr`
- Null pointers of same type are considered as equal
- Dereferencing null pointers is **undefined behavior**

```
int safe_deref(const int* x) { // just as an example
  if (x == nullptr)
    return 0;
  return *x;
}
```

[Slide 146] Null Pointers

Quiz: Which answer is NOT correct?

```
int safe_deref2(const int* x) {
  int v = *x;
  if (x == nullptr)
    return 0;
  return v;
}
```

A. The compiler can simply remove the null check.
B. The program might crash when `nullptr` is passed.

[9] https://en.cppreference.com/w/cpp/language/pointer#Null_pointers

C. The program might return zero.
D. The null check prevents an invalid pointer dereference.

> Undefined behavior might lead to seemingly surprising behavior in optimizing compilers.

[Slide 147] Subscript Operator[10]

- Treat pointer as pointer to first element of an array
- Follow the same semantics as the array subscript

```
std::array<int, 3> arr = {12, 34, 45};
const int* ptr = &arr[0]; // pointer to first element, no dereference

for (unsigned i = 0; i < 3; i++)
  std::println("{}", ptr[i]);
```

- C-style arrays often implicitly decay to pointers to the first element

```
int arr[] = {12, 34, 45};
const int* ptr = arr; // pointer to first element
```

[Slide 148] Pointer Arithmetic: Addition[11]

- `ptr + idx`/`ptr - idx`: move pointer `idx` *elements* to left/right
  - Moves underlying address by `idx * sizeof(*ptr)`
- `ptr[idx]` equals `*(ptr + idx)`; `&ptr[idx]` equals7 `ptr + idx`

```
std::array<int, 3> arr = {12, 34, 45};
const int* ptr = &arr[1]; // pointer to second element

// prints: 12 45
std::println("{} {}", *(ptr - 1), *(ptr + 1));
```

[Slide 149] Pointer Arithmetic: Past-The-End Pointers

- Only valid pointers are allowed to be dereferenced
- Pointers shall point to valid objects or be `nullptr`
- Exception: pointer past the end of the last element is allowed

⇝ Constructing out-of-bounds pointers is **undefined behavior**

```
std::array<int, 3> arr = {12, 34, 45};
const int* begin = &arr[0]; // OK, points to first element
const int* end = &arr[arr.size()]; // OK, past-the-end pointer

for (const int* p = begin; p != end; ++p) // OK
  std::println("{}", p);
```

[10] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator
[11] https://en.cppreference.com/w/cpp/language/operator_arithmetic#Additive_operators

```
int v = *end; // NOT OK: dereferencing past-the-end pointer
int* oobPtr = begin + 4; // NOT OK: pointer out of bounds
```

[Slide 150] Pointer Arithmetic: Subtraction

- Assuming two pointers `ptr1` and `ptr2` point into the same array
- `ptr1 - ptr2` is the number of elements between the pointers

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

  std::ptrdiff_t diff1 = ptr2 - ptr1; // 3
  std::ptrdiff_t diff2 = ptr1 - ptr2; // -3
}
```

[Slide 151] String Literals Quiz

Quiz: What is the output of the program?

```
#include <print>
int main() {
  std::println("{}", "Hello!" + 3);
}
```

A. (compile error) B. (undefined behavior) C. `"Hello!3"` D. `"lo!"` E. (an address)

[Slide 152] Void Pointer[12]

- Pointer to `void` is allowed
- Pointers can be implicitly converted to void pointer (retaining cv-quals)
- To use void pointer, it must be casted to a different type

- Used to pass object of unknown type
- Often used in C interfaces (e.g., `malloc`)
- Tentatively avoid in C++

[Slide 153] `static_cast`[13]

- `static_cast<new type>(expression)`
- Cast expression to "related" type, must be at least as cv-qual'ed
  - E.g., cast from void pointer to pointer of different type
  - Many more cases, see reference

```
int i = 42;
void* vp = &i; // OK, no cast required
int* ip = static_cast<int*>(vp); // OK
```

[12] https://en.cppreference.com/w/cpp/language/pointer#Pointers_to_void
[13] https://en.cppreference.com/w/cpp/language/static_cast

```
long* lp = static_cast<long*>(ip); // ERROR
long* lp = static_cast<long*>(vp); // Undefined behavior!

double d = static_cast<double>(i);
```

[Slide 154] `reinterpret_cast`[14]

- `reinterpret_cast<new type>(expression)`
- Cast expression to "unrelated" type, reinterpreting bit pattern
- Very limited set of allowed conversions
  - E.g., converting pointer to object to pointer to `char` or `std::byte`
- Invalid conversions usually lead to **undefined behavior**
- Only use when strictly required! Also avoid C-style casts

[Slide 155] Strict Aliasing Rule

- Object access with an expression of a different type is **undefined behavior**

⇒ Accessing an `int` through a `float*` is not allowed (pointer aliasing)

⇒ Compilers assume that pointers of different types have different values

- (There are few exceptions)

```
float f = 42.0f;
// Undefined behavior!
int i = *reinterpret_cast<int*>(&f);
```

[Slide 156] Pointers are Actually Complex

- Pointers generally consist of the address of the pointed-to object
- But: pointers have more semantic information (provenance[15])
  - Pointers have "information" about the underlying object
  - Used for compiler optimization
- Some hardware platforms have unusual addressing schemes
  - E.g., CHERI with 128-bit capabilities, basically pointer with bounds and permissions

[Slide 157] Pointers vs. References

	Reference	Pointer
Usable for passing-by-reference?	Yes	Yes
Guaranteed non-null?	Yes	No
Is an object itself?	No	Yes
Can change which object is referred to?	No	Yes
Supports pointer arithmetic?	No	Yes

[14] `https://en.cppreference.com/w/cpp/language/reinterpret_cast`
[15] `https://www.ralfj.de/blog/2020/12/14/provenance.html`

Recommendation (we will revisit this later):

- Prefer references for pass-by-references
- Use pointer for: optional references (`nullptr`), pointer changes object, pointer arithmetic required, storing references in an array

[Slide 158] References, Arrays, Pointers – Summary

- Value classes lvalues (locations) and rvalues
- References are aliases to other objects
- Rvalue references extend lifetime of temporary objects
- Arrays contiguously store multiple elements of same type
- String literals are a special case of an array
- Pointers are objects that point to other objects, or `nullptr`
- Pointers support arithmetic
- Pointer casts are possible, but are often invalid

[Slide 159] References, Arrays, Pointers – Questions

- Why are arrays of references impossible?
- How can the object referenced by a reference be changed?
- How to pass an object by-reference in C++?
- What is the difference between lvalue and rvalue references?
- What is different between const-lvalue and rvalue references?
- What is the relation between arrays and pointers?
- Which operations on pointers are undefined behavior?
- When is using pointer advisable over using a reference?

5. Classes and Conversions

[Slide 161] `static_assert`[1]

- `static_assert(bool expr, string)` – assert at compile-time
- Expression must be a compile-time constant
- Can have an optional failure message

Example:

```
static_assert(sizeof(int) == 4, "program␣only␣works␣on␣4-byte␣integers");
```

5.1. Classes

[Slide 162] Classes

```
class Name1 {
    // member specifications...
};
struct Name2 {
    // member specifications...
};
```

- Name can be any valid identifier
- Members can be:
  - Variables (data members)
  - Functions (member functions)
  - Types (nested types)
- Note the trailing semicolon

[Slide 163] Data Members[2]

- Declarations of (non-`extern`) variables
- Size of declared variable must be known (see later)
- Variable name must be unique within class
- Variables can have default value

```
class Name {
    int foo = 10;
    int& iref;
    float* ptr;
    const char x;
};
```

[1] https://en.cppreference.com/w/cpp/language/static_assert
[2] https://en.cppreference.com/w/cpp/language/data_members

[Slide 164] Data Layout

- Class is essentially just a sequence of its data members
- Members are stored in memory in declaration order
- Alignment of members is respected ⇝ padding between objects
- Alignment of class is largest alignment of data members

```
class C {
    int i; // sizeof = 4; alignof = 4; offset = 0
    // (4 padding bytes)
    int* p; // sizeof = 8; alignof = 8; offset = 8
    char c; // sizeof = 1; alignof = 1; offset = 16
    // (1 padding byte)
    short s; // sizeof = 2; alignof = 2; offset = 18
    // (4 padding bytes -- sizeof must be multiple of alignof)
}; // sizeof(C) = 24; alignof(C) == 8
```

> Modifying members of a class generally breaks the *Application Binary Interface (ABI)* — the interface of compiled programs. This is particularly relevant for shared libraries, where often some effort is required to prevent accidental breakage. Some libraries deliberately include unused space that they can repurpose without changing the interface visible to other translation units.
>
> Maintaining ABI stability is extremely difficult in practice and requires a high degree of programmer discipline. This is part of the reason why "modern" libraries or languages prefer static linking.

[Slide 165] Data Layout

Quiz: What is the size of `Line`?

```
class Point {
    int x;
    int y;
    unsigned char color;
};
class Line {
    Point a;
    Point b;
    unsigned char lineWidth;
};
```

A. (compile error) B. 19 C. 24 D. 28 E. 32

[Slide 166] Bit Fields[3]

- Can specify bit-size for integer members
- Adjacent bit fields packed together
- Access is fairly expensive, but might reduce memory usage

⇝ Use only when strongly beneficial

[3] `https://en.cppreference.com/w/cpp/language/bit_field`

```
class Bitfields {
    unsigned short flagA : 1;
    unsigned short flagB : 1;
    unsigned short tinyVar : 11;
};
static_assert(sizeof(Bitfields) == 2);
static_assert(alignof(Bitfields) == 2);
```

> Using bit fields is generally not recommendable. Accessing bit fields requires generation of bitwise operations to extract or insert the value, which is fairly expensive.
>
> However, when the size of the data type is particularly important, combining flags or variables with very limited ranges into bit fields can improve memory usage (and thereby performance).

[Slide 167] Data Layout

Quiz: What is the size of this class?

```
class Value { // (excerpt from llvm/include/llvm/IR/Value.h)
  const unsigned char SubclassID;
  unsigned char HasValueHandle : 1;
  unsigned char SubclassOptionalData : 7;
  unsigned short SubclassData;
  unsigned NumUserOperands : 27;
  unsigned IsUsedByMD : 1;
  unsigned HasName : 1;
  unsigned HasMetadata : 1;
  unsigned HasHungOffUses : 1;
  unsigned HasDescriptor : 1;
  Type *VTy;
  Use *UseList;
}; // NB: sizeof(void*) == 8; sizeof(unsigned) == 4
```

A. (compile error) B. 24 C. 32 D. 40 E. 45

[Slide 168] Data Layout: Consequences

- Order of members has impact on class size

⇒ When class size is important, reduce padding

⇒ Recommendation: place all data members together at beginning/end

  - Potential padding etc. is easily findable

- All users of the class need to know the declaration

⇒ Class declarations often put in header files

⇒ Adding/modifying members requires changes data layout ⇒ recompilation

  - Especially important when distributing libraries – all users *must* rebuild

[Slide 169] Member Functions

- Declaration of methods just like regular function declarations

- Inline definitions are implicitly `inline`
- Out-of-line definitions are preferable for non-trivial methods

```
//--- foo.h
#pragma once
class Foo {
    int foo();
    int bar(int x) { // inline definition
        return x + 1;
    }
};
//--- foo.cpp
int Foo::foo() { // out-of-line definition
    return 10;
}
```

> Inline definitions are typically preferable, when the function is very small and giving the compiler the possibility of inlining the function gives a substantial performance benefit. For example, inlining a simple get-function for a data member is typically preferable. For more complex logic, out-of-line definitions in `.cpp`-files are generally preferable (why?).

[Slide 170] Inline vs. Out-Of-Line Definitions

Quiz: Which answer is NOT correct?

A. Out-of-line definitions tend to allow for more optimizations.
B. Out-of-line definitions tend to reduce compile time.
C. Inline definitions tend to allow for more optimizations.
D. Inline definitions in headers are possibly compiled several times.

- Similar considerations as for `inline` functions apply

[Slide 171] Member Access

```
struct Vec {
    unsigned x;
    unsigned y;
};
Vec v;
Vec* vp = ...;

// member access:
int l1dist_a = v.x + v.y;
// ptr->member is a shorthand for (*ptr).member
int l1dist_b = vp->x + vp->y;
```

[Slide 172] `this`

- Member functions have implicit parameter `this`; type is `Class*`
- In member functions, members can be accessed without `this` (preferred)

```
struct Vec {
    unsigned x;
    unsigned y;

    unsigned l1dist() {
        return this->x /* explicit access */ + y /* implicit access*/;
    }
};
Vec v;
Vec* vp = ...;
int l1dist_a = v.l1dist();
int l1dist_b = vp->l1dist();
```

[Slide 173] `const`-Qualified Member Functions

- Member functions can be `const`-qualified
- `this` is a `const Class*`

⇒ Data members are immutable

```
struct Vec {
    unsigned x;
    unsigned y;
    unsigned getX() const { return x; }
    unsigned getY() const { return y; }
    unsigned l1dist() const;
};
unsigned Vec::l1dist() const {
    return x + y; // this is a const Vec*
}
```

> When a member function does not modify the object, it is highly recommended to `const`-qualify it.

[Slide 174] Constness and Member Functions

- For *non-const lvalues* non-const overloads are preferred over const ones
- For *const lvalues* only const-qualified functions are selected

```
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 = /* ... */;
```

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

[Slide 175] Constness of Member Variables

- Constness propagates through pointer lvalue access
- `const` data members are always constant
  - Can only be set once during construction (see later)
- `mutable` member variables are always non-const (use carefully!)

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

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

[Slide 176] Static Members[4]

- Static data members: members not bound to class instances
- Only one instance in the program, like global variables
- Static member functions: no implicit `this` parameter
- Static members can be accessed with `::` operator

```
//--- foo.h
struct Foo {
    static int var; // declaration
    static void statfn(); // declaration
};
//--- foo.cpp
int Foo::var = 10; // definition
```

[4] https://en.cppreference.com/w/cpp/language/static

```
void Foo::statfn() { /* ... */ } // definition
```

> Note that static data members must have an out-of-line definition and, as global variables, must be defined exactly once in the entire program.

5.2. Constructors

[Slide 177] Constructors

- ... are special functions that are called when an object is *initialized*
- ... have no return type, no const-qualifier, and name is class name
- ... can have arguments, constructor without arguments is *default constructor*
- ... are sometimes implicitly defined by the compiler

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

[Slide 178] Initializer List

- Specify how member variables are initialized before constructor body
- Other constructors can be called in the initializer list
- Members initialized in the order of their definition
- `const` member variables can only be initialized in the initializer list

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

[Slide 179] Initializing Objects[5]

- Constructor executed on initialization

[5] https://en.cppreference.com/w/cpp/language/initialization

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

[Slide 180] Constructors (1)

Quiz: What is the output of the following program?

```
#include <print>
struct Foo {
    int answer;
    Foo() : answer(42) {}
};
int main() {
    Foo f();
    std::println("{}", f.answer);
    return 0;
}
```

A. (compile error) B. 0 C. 42 D. (undefined behavior)

[Slide 181] Constructors (2)

Quiz: What is the return value of `foo`?

```
struct C {
    int i;
    C() = default;
};
int foo() {
    const C c;
    return c.i;
}
```

A. (compile error) B. an arbitrary integer C. 0 D. (undefined behavior)

[Slide 182] Constructors (3)

Quiz: What is problematic about this program?

```
#include <print>
struct Foo {
    const int& answer;
    Foo() {}
```

```
    Foo(const int& answer)
        : answer(answer) {}
};
int main() {
    int answer = 42;
    Foo f(answer);
    std::println("{}", f.answer);
    return 0;
}
```

A. Compile error: Two constructors are not allowed.
B. Compile error: `answer` not always initialized.
C. Compile error: `f` is a function declaration.
D. Undefined behavior: `f.answer` is a dangling reference.
E. There is no problem: the program always prints 42.

[Slide 183] Constructors (4)

Quiz: What is problematic about this program?

```
#include <print>
struct Foo {
    const int& answer;
    Foo(const int& answer)
        : answer(answer) {}
};
int main() {
    int answer = 42;
    Foo f = answer;
    std::println("{}", f.answer);
    return 0;
}
```

A. Compile error: Cannot assign integer to type `Foo`.
B. Compile error: Cannot convert integer to `Foo`.
C. Undefined behavior
D. There is no problem: the program always prints 42.

[Slide 184] Converting and Explicit Constructors[6]

- Constructors with one argument used for *explicit* and *implicit conversions*
- Use `explicit` to disallow implicit conversion
- Generally, use `explicit` unless there's a good reason not to

```
struct Foo {
    Foo(int i);
};
void print_foo(Foo f);
// Implicit conversion,
// calls Foo::Foo(int)
```

[6] https://en.cppreference.com/w/cpp/language/converting_constructor

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

5.3. Member Access Control

[Slide 185] Member Access Control

- Every member has `public`, `protected` or `private` access
- Default for `class`: `private`; for `struct`: `public`
  - Recommendation: always explicitly specify access control
- `public` = accessible by everyone, `private` only by class itself

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

[Slide 186] Friend Declarations[7]

- Class body can contain *friend declarations*
- Friend: has access to private/protected members
- `friend function-declaration;` (for friend function)
- `friend class-specifier;` (for friend class)

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

[7] https://en.cppreference.com/w/cpp/language/friend

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

[Slide 187] Nested Types

- For nested types classes behave just like a namespace
- Nested types are accessed with `::`
- Nested types are `friend`s of their parent

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

5.4. Forward Declarations

[Slide 188] Forward Declarations

- Classes can be forward declared: `class Name;`
- Type is *incomplete* until defined later
- Incomplete type can be used, e.g., for pointer/reference

```
//--- foo.h
class A;
class ClassFromExpensiveHeader;
class B {
    ClassFromExpensiveHeader* member;
    void foo(A& a);
};
class A {
    void foo(B& b);
};
//--- foo.cpp
#include "ExpensiveHeader.hpp"
// ...
```

> There are two benefits of forward-declaring classes: Inclusion of some headers can be avoided, leading to faster build times, and mutual references between classes become

possible.

Thus, forward declarations are sometimes used in header files, when the actual class definition is not required.

[Slide 189] Incomplete Types[8]

- No operations that require size/layout of type are possible
  - No pointer arithmetic
  - No access to members, member functions, etc.
  - No definition/call of function with incomplete return/argument type
- Sometimes, this information is not needed:
  - E.g., pointer/reference declarations can refer to incomplete types
  - E.g., member functions with incomplete parameter types

5.5. Operator Overloading

[Slide 190] Operator Overloading[9]

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

[Slide 191] Arithmetic Operators[10]

lhs op rhs ∼ *lhs*`.operator` *op*`(`*rhs*`)` or `operator` *op*`(`*lhs*`,` *rhs*`)`

- Overloaded versions of `||` and `&&` lose their special behaviors
- Should be `const` and take `const` references
- Usually return a value and not a reference

```
struct Int {
    int i;
    Int operator+(const Int& other) const { return Int{i + other.i}; }
    Int operator-() const { return Int{-i}; };
};
Int operator*(const Int& a, const Int& b) { return Int{a.i * b.i}; }
Int a{123}; Int b{456};
a + b; /* is equivalent to */ a.operator+(b);
a * b; /* is equivalent to */ operator*(a, b);
-a; /* is equivalent to */ a.operator-();
```

[8] https://en.cppreference.com/w/cpp/language/types#Incomplete_type
[9] https://en.cppreference.com/w/cpp/language/operators
[10] https://en.cppreference.com/w/cpp/language/operator_arithmetic

[Slide 192] Comparison Operators[11]

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

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

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

[Slide 193] Three-Way[12]

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

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

[Slide 194] Three-Way Comparison (2)

- `partial_ordering` can be unordered, i.e. neither `a <= b` nor `a >= b`
  - `std::partial_ordering::less`, `::equivalent`, `::greater`, `::unordered`
  - Example: floating-point numbers, NaN is unordered
- `std::weak_ordering` or `std::strong_ordering` for total order
  - `::less`, `::equivalent`, `::greater`
  - `strong_ordering`: equal values must be completely indistinguishable
  - Example for strong ordering: integers
  - Example for weak ordering: points in 2d-space ordered by distance from origin

[Slide 195] Increment and Decrement[13]

Pre- and post-inc/dec are distinguished by an (unused) `int` argument

[11] `https://en.cppreference.com/w/cpp/language/operator_comparison`
[12] `https://en.cppreference.com/w/cpp/utility/compare/partial_ordering`
[13] `https://en.cppreference.com/w/cpp/language/operator_incdec`

- `C& operator++(); C& operator--();` pre-increment or -decrement, modify object, return `*this`
- `C operator++(int); C operator--(int);` post-increment or -decrement, copy self, modify self, return unmodified copy

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

[Slide 196] Subscript Operator[14]

Classes behaving like containers/pointers usually override the *subscript* `[]`

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

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

[Slide 197] Dereference Operators[15]

Classes behaving like pointers usually override the operators `*` and `->`

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

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

[Slide 198] Assignment Operators[16]

- Operator = is often used for copying/moving (see next week)
- All assignment operators usually return `*this`

[14] https://en.cppreference.com/w/cpp/language/operator_member_access
[15] https://en.cppreference.com/w/cpp/language/operator_member_access
[16] https://en.cppreference.com/w/cpp/language/operator_assignment

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

[Slide 199] Conversion Operators[17]

- Conversion can be done using converting constructors (seen before)
- or *conversion operators*: `operator type ()`

- The `explicit` keyword can be used to prevent implicit conversions
- Explicit conversions are done with `static_cast`

```
struct Int {
    int i;
    operator int() const {
        return i;
    }
};
Int a{123};
int x = a; // OK, x is 123
struct Float {
    float f;
    explicit operator float() const {
        return f;
    }
};
Float b{1.0};
float y = b; // ERROR, implicit conversion
float y = static_cast<float>(b); // OK
```

[Slide 200] `operator bool`

- `operator bool`: converts to `bool`
- Used to enable use of object in `if`, `while`, etc.
  - `if`, `while` etc. perform an *explicit* conversion

```
struct Ptr {
    void *p;
    explicit operator bool() const {
        return p; // pointers have an implicit conversion to bool
    }
};
Ptr p{nullptr};
if (p) {} // OK: explicit conversion
bool hasPtr = p; // ERROR: implicit conversion
```

[17] https://en.cppreference.com/w/cpp/language/cast_operator

[Slide 201] Argument-Dependent Lookup[18]

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

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

5.6. Enums

[Slide 202] Enums[19]

- Typically used like integral types with a restricted range of values
- Also used to assign descriptive names instead of "magic" integer values
- Syntax: *enum-key name* `{` *enum-list* `};`
- *enum-key* can be `enum`, `enum class`, or `enum struct`
- Without explicit value, first element gets zero, other increment from previous

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

[Slide 203] Using Enum Values

- Names from the enum list can be accessed with the scope resolution operator
- Enums can be converted to integers and vice versa with `static_cast`
- `enum` without `class`/`struct`: C-style enums
  - Names also introduced in the enclosing namespace
  - Can be converted implicitly `int`
- `enum class` and `enum struct` are equivalent
- Recommendation: Use `enum class` unless you have a reason not to

[18] https://en.cppreference.com/w/cpp/language/adl
[19] https://en.cppreference.com/w/cpp/language/enum

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

5.7. Type Aliases

[Slide 204] Type Aliases[20]

- Type names nested deeply in namespaces/classes can become very long

⇝ *Type alias*: `using` |*name*| `=` |*type*|`;`

- *name* is the name of the alias, *type* must be an existing type
- (C compatibility: equivalent to `typedef`, but prefer `using`)

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

> Type aliases in classes are already somewhat useful know, as they allow to change the concrete type at only one location, e.g., the integer type used for storing indices in a container.
>
> Later with templated classes, nested type aliases become much more important, as they allow for a unified interface to access, for example, the element type of a list without further knowledge on the list type itself.

[Slide 205] Classes and Conversions – Summary

- Classes are a sequence of their data members
- Classes can have member functions with implicit `this` pointer
- Member functions can be `const-qualified`
- Constructors are called for initializing objects
- Constructors and operators provide implicit/explicit conversions
- Class members can have different access control
- Access control can be circumvented by `friend` declarations
- Almost all operators can be overloaded with custom semantics
- Enums are, optionally scoped, integer types with descriptive value names

[Slide 206] Classes and Conversions – Questions

- What is the difference between `class` and `struct`?
- When is padding required between fields?
- How can the size of a struct be reduced?

[20] `https://en.cppreference.com/w/cpp/language/type_alias`

- What is the type of `this`? Is it always the same?
- Why do methods returning references typically have a non-const-qualified and a const-qualified overload? Which overload is taken in which cases?
- Why do references members have to be initialized in initializer lists?
- Why could massive operator overloading be problematic in large projects?
- How to access the raw integer value of `enum class` enumerators?

6. Memory Management and Copy/Move

6.1. Heap Allocations

[Slide 208] Stack vs. Heap Memory

Stack Memory

- Used for objects with automatic storage duration
- Compiler can decide when allocation/dealloc happens

\+ Fast allocation/deallocation
– No dynamic data structures
– Only small allocations (few kiB)
– Memory freed on return

Heap Memory

- Used for objects with dynamic storage duration
- Programmer explicitly manages allocation/deallocation

\+ Very flexible
– Alloc/dealloc is expensive
– Memory fragmentation
– Error prone!
  – Memory leaks, double free

[Slide 209] Dynamic Memory Management

- Create and initialize object: `new` *`type initializer`*[1]
  – Type must be a type, can be an array; initializer optional
  – Allocates heap storage, initializes object, returns pointer
- Destroy object and release memory: `delete` *`expr`*/`delete[]` *`expr`*[2]
  – Expression must be a pointer allocated by `new`; ignored if `nullptr`
  – Invoke destructor, deallocate memory

> There are a few things to consider when using `new`/`delete`:
>
> - `new`/`delete` form must match, an array allocated with `new type[]` must be destroyed through `delete[]`. Furthermore, it is not allowed (or even advisable) to mix C's `malloc` with `delete` or `new` with C's `free`.
>   (If were to use `malloc`, you would need to construct an object via *placement new* before using it, e.g. `new (mallocBuffer) Type()`; and then manually call the destructor before `free`, e.g. `ptr->~Type();`. Not doing so is undefined behavior (object used outside its lifetime). This is *strongly* discouraged unless strictly required.)
> - Not destroying an allocated object is permitted ("memory leak"), but can be

[1] `https://en.cppreference.com/w/cpp/language/new`
[2] `https://en.cppreference.com/w/cpp/language/delete`

very problematic. It is easy to miss a missing `delete`.
- Destroying an object multiple times is not just undefined behavior, but also a way to introduce security vulnerabilities.

Thus, it is strongly recommended to *not* use `new`/`delete` and instead use smart pointers with ownership semantics to prevent (and/or reduce the risk of) many of these problems.

[Slide 210] `new`/`delete` Example

```
#include <print>
class Foo {
    const int birthYear;
public:
    explicit Foo(int birthYear) : birthYear(birthYear) {}
    int getAge(int year) const { return year - birthYear; }
};

int main() {
    Foo* foo = new Foo(2021);
    std::println("age:␣{}", foo->getAge(2024));
    delete foo;
    return 0;
}
```

[Slide 211] `new`/`delete` Example

Quiz: What is the output of the program?

```
#include <print>
#include <string_view>
class Ballot {
    const bool voteGOP;
public:
    Ballot(bool voteGOP) : voteGOP(voteGOP) {}
    std::string_view getParty() const { return voteGOP ? "GOP" : "DNC"; }
};
int main() {
    Ballot ballot = new Ballot(/*voteGOP=*/false);
    std::println("Voted␣{}", ballot.getParty());
    return 0;
}
```

A. (compile error) B. `Voted DNC` C. `Voted GOP` D. (undefined behavior)

[Slide 212] `new`/`delete` Example

Quiz: What is problematic about this function?

```
#include <ctime>
class Ballot { /* ... */ };
Ballot* castBallot() {
```

```
    std::time_t time = std::time(nullptr); // UNIX timestamp
    Ballot* ballot = new Ballot(time % 2); // Informed decision
    if (time % 5 == 3)
        return nullptr; // ... polling station is too far away :(
    return ballot;
}
```

A. Memory is leaked when the condition is taken.
B. Memory is leaked when the condition is not taken.
C. Memory is always leaked.
D. The function does not always return the same value.

6.2. Destructor

[Slide 213] Destructor[3]

- Special function called when lifetime of object ends
  - For automatic storage dur: called at scope end in reverse definition order
  - Destructors of class members called automatically in reverse order
- No return time, no arguments, name `~ClassName()`
- Typical use: deallocate managed resources

[Slide 214] Destructor: Example

```
struct Bar { /* ... */ };
struct Foo {
    Bar b1;
    Bar b2;
    ~Foo() {
        std::println("bye");
        // b2.~Bar(); called
        // b1.~Bar(); called
    }
};
void doFoo() {
    Foo f;
    { Bar b; /* b.~Bar(); called */ }
    // f.~Foo(); called
}
```

[Slide 215] Using Destructors to Deallocate Resources

```
class FooPtr {
    Foo* ptr;
public:
    FooPtr(int birthYear) : ptr(new Foo(birthYear)) {
        std::println("new␣{}", static_cast<void*>(ptr));
    }
    ~FooPtr() {
```

[3] https://en.cppreference.com/w/cpp/language/destructor

```
        std::println("deleted␣{}", static_cast<void*>(ptr));
        delete ptr;
    }
    Foo& operator*() const { return *ptr; }
    Foo* operator->() const { return ptr; }
};
int main() {
    FooPtr foo(2021);
    std::println("age:␣{}", foo->getAge(2024));
    return 0;
}
```

The `FooPtr` wrapper automatically destroys the heap-allocated object when it goes out of scope.

[Slide 216] `new`/`delete` Example

Quiz: What is problematic about this code?

```
class FooPtr { /* ... */ };
void printAge(FooPtr foo) {
    std::println("age:␣{}", foo->getAge(2024));
}
int main() {
    FooPtr foo(2021);
    printAge(foo);
    return 0;
}
```

A. An instance of `Foo` is leaked.
B. The `getAge` call uses an object outside its lifetime.
C. The same instance of `Foo` is destroyed twice.
D. There is no problem.

6.3. Copy Semantics

[Slide 217] Copy Semantics

- Assignment/construction typically copies object
- By default, copy is *shallow*
- Ok for fundamental types, problematic for user-defined types
- Copying may be expensive
- Copying may be unintended/avoidable
- **Copying is problematic with managed resources**
  - Might cause leak, when assigned-to object already has resources
  - Might cause double-free

[Slide 218] Copy Constructor[4]

- Syntax: `ClassName(const ClassName&)`
- Invoked on initialization from an object of same type:
  - Copy initialization: `T a = b;`
  - Direct initialization: `T a(b);`
  - Argument passing: `f(b)` for `void f(T a);`

```
class FooPtr {
    // ...
    FooPtr(const FooPtr& other) : ptr(new Foo(*other)) {}
    //...
};
```

[Slide 219] Copy Assignment[5]

- Syntax 1: `ClassName& operator=(const ClassName&)` (preferred)
- Syntax 2: `ClassName& operator=(ClassName)` (sometime useful, see later)
- Typically returns `*this`
- Invoked when assigning to an already initialized object
  - `a = b;`

```
class FooPtr {
    // ...
    FooPtr& operator=(const FooPtr& other) { // PROBLEMATIC, see next slide
        delete ptr;
        ptr = new Foo(*other);
        return *this;
    }
    //...
};
```

[Slide 220] Copy Assignment

Quiz: What is problematic about this code?

```
class FooPtr { /* ... */
    FooPtr& operator=(const FooPtr& other) {
        delete ptr; ptr = new Foo(*other);
        return *this;
    } /* ... */ };
int main() {
    FooPtr foo(2021);
    foo = foo;
}
```

A. Some memory is used after it has been freed.
B. The `delete`/`new` is unnecessary.

[4] https://en.cppreference.com/w/cpp/language/copy_constructor
[5] https://en.cppreference.com/w/cpp/language/copy_assignment

C. Self-assignment of classes is forbidden in C++.
D. There is no problem.

[Slide 221] Copy Assignment (fixed)

```
class FooPtr {
    // ...
    FooPtr& operator=(const FooPtr& other) { // Fixed version
        if (this == &other) // check for self-assignment
            return *this;

        delete ptr; // NB: could try to reuse storage
        ptr = new Foo(*other);
        return *this;
    }
    //...
};
```

[Slide 222] Implicit Declaration of Copy Constructor/Assignment

- Compiler implicitly declares copy constructor/assignment if not explicitly declared
  - Will be `public inline` and perform member-wise copy in initialization order
- Implicit copy constructor/assignment *deleted*, if:
  - Class has members that cannot be copy-constructed/assigned; or
  - Class has a user-defined move constructor or assignment operator
- See reference for more details
- Explicit deletion: `T(const T&) = delete;`
- Explicit deletion: `T& operator=(const T&) = delete;`

[Slide 223] Custom Copy Operations: Guidelines

- If implicit copy not sufficient: typically should not be copyable
- Exception: if class manages resources, e.g. dynamic memory
- **Rule of three**[6]: If one of the following is user-defined, all three have to be: destructor, copy constructor, copy assignment
  - Custom destructor: cleanup needs to be done on copy assignment
  - Custom copy constructor: custom setup, needs to be done in copy assignment
  - Custom resource management (e.g., file descriptor): implicit versions incorrect

6.4. Move Semantics

[Slide 224] Move Semantics

- Copy semantics often incur avoidable overhead
- Object might be immediately destroyed after copy

[6] `https://en.cppreference.com/w/cpp/language/rule_of_three`

- Object might be unable to share resources for copy
- Move constructor/assignment "steals" resources of argument
- Leave argument in valid, empty state (destructor will be called nonetheless)
- Indicated by use of rvalue reference

> The intution is: rvalue references indicate references to temporaries that expire when the reference goes out of scope, i.e., for function parameters the value's lifetime ends at the end of the function anyway — thus, just take the resources before they are destroyed.

[Slide 225] Move Constructor[7]

- Syntax: `ClassName(ClassName&&) noexcept`
- Invoked on initialization from an temporary value of same type

⇝ Steal resources of argument, its lifetime ends at the constructor end

```
class FooPtr {
    // ...
    FooPtr(FooPtr&& other) : ptr(other.ptr) {
        other.ptr = nullptr; // Must leave in valid, empty state for destructor
    }
    //...
};
```

[Slide 226] Move Assignment[8]

- Syntax: `ClassName& operator=(ClassName&&) noexcept`
- Invoked when assigning an rvalue to an already initialized object
  - `a = b();`

```
class FooPtr {
    // ...
    FooPtr& operator=(FooPtr&& other) {
        if (this == &other)
            return *this;
        delete ptr;
        ptr = other.ptr;
        other.ptr = nullptr;
        return *this;
    }
    //...
};
```

[Slide 227] Implicit Declaration of Move Constructor/Assignment

- Compiler implicitly declares move constructor/assignment if:
  - No user-declared copy/move constructors, assignment operators, and destructors

[7] https://en.cppreference.com/w/cpp/language/move_constructor
[8] https://en.cppreference.com/w/cpp/language/move_assignment

- Implicit move constructor/assignment *deleted*, if:
  - Class has members that cannot be move-constructed/assigned; or
  - Class has member of reference type
- See reference for more details
- Explicit deletion possible similar to copy constructor/assignment

[Slide 228] Custom Move Operations: Guidelines

- If class manages resources: custom move often necessary
- Move operations should not allocate new resources
- Moved-from object must remain in valid state
- **Rule of five**[9]:
  - If move semantics are desired: need all five special member functions
  - If only move semantics are desired: still need all five, define copy as deleted
- Implementing move operations is typically a pure optimization

[Slide 229] Converting Lvalue to Rvalue Reference

- Want to move object?
- But only have an lvalue?
- `static_cast<Type&&>(obj)`
  - New value category: xvalue — eXpiring object whose resources can be reused
  - Like lvalue, object has an identity
  - Like rvalue, can be moved from (i.e., overload resolution selects rvalue-ref variant)
- Syntactic sugar (preferred): `std::move(obj)` from `<utility>`

[Slide 230] Copy/Move Constructor

Quiz: Which methods on FooPtr are called?

Assume that FooPtr implements all copy/move constructors/assignments.

```
FooPtr createFoo() { return FooPtr(2020); }
void printAge(FooPtr foo) {
    std::println("age: {}", foo->getAge(2024));
}
int main() {
    FooPtr f = createFoo();
    printAge(createFoo());
}
```

A. constr; copy-constr; destr; constr; copy-constr; destr; destr; destr
B. constr; copy-constr; destr; constr; move-constr; destr; destr; destr
C. constr; constr; copy-constr; destr; destr; destr

[9] https://en.cppreference.com/w/cpp/language/rule_of_three#Rule_of_five

D. constr; constr; destr; destr

[Slide 231] Copy Elision[10]

- Compilers (must) sometimes omit copy/move constructors if object can be directly in storage where it would be copied/moved to
- Examples: return values, arguments with prvalue

⇒ Zero-copy pass-by value semantics

- Some elisions are required by C++17, but not all

⇝ Portable programs should not rely on side-effects of constructors/destructor

6.5. Idioms

[Slide 232] Copy-And-Swap

- Class defines `ClassName& operator=(ClassName)` for copy/move
- Exchange resources between argument and `*this`
- Copy constructor creates copy
- Let destructor clean up resources at function return

```
class FooPtr {
    Foo* ptr;
public:
    ~FooPtr() { delete ptr; }
    FooPtr(const FooPtr& other) : ptr(new Foo(*other)) {}
    FooPtr& operator=(FooPtr other) {
        std::swap(ptr, other.ptr);
        return *this;
    } // destructor of other cleans up formerly own resources
};
```

> Note that using this idiom might temporarily use more resources than strictly required: On a copy assignment, a new copy is made unconditionally and existing resources (e.g., memory) are not reused.

[Slide 233] Resource Acquisition is Initialization (RAII)[11]

- Idea: bind lifetime of resource to lifetime of an object
  - Resources: heap memory, files, mutex, database connection, ...

⇒ Guarantees resource availability during lifetime of object

⇒ Guarantees that resource is released at lifetime end of object

- Encapsulate each resource into a class solely responsible for managing it
- Constructor acquires resource; destructor releases resource
- Delete copy ops, implement custom move ops

[10] https://en.cppreference.com/w/cpp/language/copy_elision
[11] https://en.cppreference.com/w/cpp/language/raii

[Slide 234] RAII Example

```
class FooPtr {
    Foo* ptr;
public:
    FooPtr(int birthYear) : ptr(new Foo(birthYear)) {}
    ~FooPtr() { delete ptr; }
    FooPtr(const FooPtr& other) = delete;
    FooPtr(FooPtr&& other) : ptr(other.ptr) { other.ptr = nullptr; }
    FooPtr& operator=(const FooPtr& other) = delete;
    FooPtr& operator=(FooPtr&& other) { // code style condensed for slide :|
        if (this != &other) { delete ptr; ptr = other.ptr; other.ptr = nullptr; }
        return *this;
    }
};
int consumeFoo(FooPtr foo) {
    if (condition)
        return 1; // No need to free memory
    // ...
    return 0;
}
int main() {
    FooPtr foo(2020);
    return consumeFoo(std::move(foo)); // foo is empty now
}
```

[Slide 235] RAII: Implications

- One of the most important and powerful idioms in C++
- RAII objects should have automatic(/temporary) storage duration
  - ⇝ Compiler manages lifetime and thus resource
- Don't use `new`/`delete` outside of RAII class
  - C++ provides *smart pointers* for this, see later
- Keep RAII classes (custom copy/move/destructor) small and focused
- For all other classes, use default or delete
  - Rule of zero[12]

6.6. Ownership

[Slide 236] Ownership Semantics

- Enabled by RAII idiom with move semantics
- A resource is always "owned", i.e., encapsulated by exactly one C++ object
- Ownership can be transferred by moving the object
  - Pass RAII class by value or return to indicate transfer of ownership
- *Very rarely*, shared ownership is needed

[12] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero

[Slide 237] `std::unique_ptr`[13]

- Smart pointer ownership for an arbitrary pointer/array (can be `nullptr`)
- Automatically destroys object when `unique_ptr` goes out of scope
- Can be used like a raw pointer — but only moveable, not copyable
- Pass `std::unique_ptr` *by value*, not by reference
- **Prefer `std::unqiue_ptr` over raw pointers**

```
#include <memory>
class Foo { /* ... */ };
int main() {
    // make_unique forwards arguments to constructor
    std::unique_ptr<Foo> foo = std::make_unique<Foo>(2020);
    if (!foo) return 1; // contextually convertible to bool, like raw pointer
    foo->printAge(2024); // ->, * work as for raw pointers
    Foo* fp = foo.get(); // get raw pointer
    // Foo* fp2 = foo.release(); // release ownership; must delete manually
}
```

[Slide 238] `std::unique_ptr` for Arrays

- Can also be used for heap-based arrays

```
std::unique_ptr<int[]> foo(unsigned size) {
    std::unique_ptr<int[]> buffer = std::make_unique<int[]>(size);
    for (unsigned i = 0; i < size; ++i)
        buffer[i] = i;
    return buffer; // transfer ownership to caller
}
int main() {
    std::unique_ptr<int[]> buffer = foo(42);
    // do something
}
```

[Slide 239] `std::shared_ptr`[14]

- Smart pointer with shared ownership
- Resource released when last owner releases it
- Implemented through atomic reference counting
- Can be copied and moved
- Use `std::make_shared` for creation

- `std::shared_ptr` is expensive and **should be avoided where possible**

[Slide 240] `std::shared_ptr` – Example

```
#include <memory>
#include <vector>
struct Node {
    std::vector<std::shared_ptr<Node>> children;
```

[13] https://en.cppreference.com/w/cpp/memory/unique_ptr
[14] https://en.cppreference.com/w/cpp/memory/shared_ptr

```
    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
}
```

> This is not a really good example: alternatively, there could be a `struct Graph` which owns all the nodes; and the nodes then use raw pointers to reference each other. This alternative design would be more efficient and therefore often preferable.

6.7. Usage Guidelines

[Slide 241] Usage Guidelines

Param. Type	Type Copyable	Type not Copyable
`T`	Copy, small objects only	Transfer ownership
`T&`/`const T&`	No ownership transfer, object larger than pointer; `const` if callee should not modify object; don't use for `unique_ptr`&friends	
`T*`/`const T*`	Like `&`, but nullable	
`T&&`	Ownership transfer	— (use `T`)

[Slide 242] Memory Management and Copy/Move – Summary

- Heap memory manually managed with `new`/`delete`
- Classes have destructors executed at end of lifetime
- Custom copy constructor and assignment required for resource management
  - Rule of three: if you need one, you need all: destructor, copy constructor/assignment
- Custom move constructor and assignment possible as optimization
- Rvalue references indicate moving, use `std::move` for moving lvalues
- Use small RAII classes for managing resources
- Use `std::unique_ptr` instead of manual `new`/`delete`
- For shared ownership use `std::shared_ptr`, but avoid if possible

[Slide 243] Memory Management and Copy/Move – Questions

- What are problems of manually using `new`/`delete`?
- What is the difference between copy constructor and assignment?

- Why do assignment operators often guard against self-assignment?
- When are the implicitly declared constructors/assignments sufficient?
- What is the difference between copying and moving a value?
- Why pass-by-value unproblematic for returns but not for parameters?
- What does `std::move` do?
- What is the benefit of using dedicated resource/RAII classes?
- How to express ownership transfer in parameters?

7. Templates

[Slide 245] On Complexity

One of the most important ways to deal with complexity is to
leave it out.

Simple solutions are often better.

[Slide 246] Type-Generic Functionality

- Some functionality is independent of specific type
  - E.g., `std::swap`, `std::unique_ptr`, ...
- Copy-paste: massive code duplication, not maintainable
- Macros: only textual replacement ⇝ very limited

7.1. Basics

[Slide 247] Templates

- Template defines family of classes/functions/type aliases/variables
- *Compile-time* parameterization on types or constants
  - On every instantiation, code compiled with respective specialization

```
template <class T, size_t N>
class array {
  T data[N];
public:
  T& operator[](size_t i) { return data[i]; }
  // ...
};
int main() {
  array<int, 12> a1; // T substituted with int; N with 12
  array<char*, 1> a2; // T substituted with char*; N with 1
}
```

[Slide 248] Template Syntax[1]

- `template <`*parameter-list*`>` `declaration`
- Parameter list: comma-separated list of template parameters
  - `class` *name* (or `typename` *name*): Type parameter

[1] `https://en.cppreference.com/w/cpp/language/templates`

  - *type name*: Non-type parameter (int, pointer, enum, lvalue reference)
  - `template <`*parameter-list*`> class` *name*: template parameter (avoid)
- Can have default parameters (similar to functions)
- Declaration:
  - `class` or `struct` (class template)
  - A function or member function (function template)
  - `using` (alias template)
  - Variable declararion (variable template)

> Template template parameters allow to specialize templates on templated types. Example:
> ```
> template <template <class> Container> void foo() {
>   Container<int> intContainer;
>   Container<long> longContainer;
> }
> // Callable, e.g., as foo<std::vector>();
> ```
> This is needed extremely rarely and should be avoided.

[Slide 249] Using Templates

- *template-name* `<`*parameters*`>`
- Results in *specialization* of template
- Sometimes, arguments can be *deduced* automatically by compiler

```
class A; // forward declaration => incomplete type
template <class T1, class T2 = unsigned, unsigned N = 4u>
class Foo { /* ... */ };

Foo<A, int, 32u> foo1;
Foo<long, A*, 12> foo2; // 12 converted implicitly to unsigned
Foo<char> foo3;
Foo<Foo<char>**, long&> foo4;
```

> Template arguments don't need to be complete if the template itself doesn't require the type to be complete (e.g., when only using a `T1*`).

[Slide 250] More Template Examples

```
template <class T>
using Storage = std::vector<T>;

template <class T>
void swap(T& a, T& b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
int f() {
  int x = 10, y = 12;
  swap<int>(x, y);
```

```
  swap(x, y); // OK, template arguments are deduced
  return x; // returns 10
}
```

[Slide 251] More Template Examples

```
#include <array>
#include <cstddef>
struct Foo {
    template <class T>
    using ArrayType = std::array<T, 42>;

    template <class T>
    std::size_t getSize() {
        return sizeof(T);
    }
};

int f() {
    Foo::ArrayType<int> intArray;
    Foo foo;
    return foo.getSize<Foo::ArrayType<int>>();
}
```

[Slide 252] Template Instantiation

- Function/class template itself is *not* a type/object/function
  - No machine code generated from template definition!
- Template must be *instantiated*
  - Compiler generates actual function/class for a specialization
  - Specializaton ≈ smart code duplication
- Explicit instantiation: explicit request
- Implicit instantiation: specialization used in context that requires complete type

[Slide 253] Explicit Instantiation

- Force instantiation of a template specialization
- `template class` *`template-name <arguments>`*`;`
- `template` *`ret-type name <arguments>(func parameters)`*`;`
- Declaration with preceding `extern`; explicit instantiations follow ODR

```
//--- A.h
template <class T>
void reallyBigA(T* t);
extern template void reallyBigA<int>(int*);

//--- A.cpp
template <class T>
void reallyBigA(T* t) { /* ... */ }
// Causes compiler to instantiate and therefore actually compile the function
template void reallyBigA<int>(int*);
```

> The idea is that — as "usual" — the header file only provides the declaration while the definition and actual implementation is in a separate source code file.

[Slide 254] Implicit Instantiation

- Occurs when specialization is used where complete type is required
- Members of class template are only instantiated when actually used
- Definitions must be visible for instantiation ⇝ usually provided in header

```
template <class T>
struct A {
    T foo(T value) { return value + 42; }
    T bar();
};
int main() {
    A<int> a; // Instantiates only A<int>
    int x = a.foo(32); // Instantiates A<int>::foo
    // No error although A::bar is never defined!
    A<float>* aptr; // Does not instantiate A<float>
}
```

[Slide 255] Explicit vs. Implicit Instantiation

Explicit Instantiation

- \+ Definition encapsulated in .cpp file
- \+ Can be shipped as library
- − Severely limits usability

Implicit Instantiation

- \+ Flexibly usable with any type
- − Definition must be in header
- − User of templates has to compile them

- Instantiations generated locally in each compilation unit
  - − Templates are implicitly `inline`
- Compiler generates code for *each* instantiation
- Substantially increases compile time
- Usually: implicit instantiation due to flexibility

> As an example, assume that compiling the instantiation of one function template leads to 1 kiB machine code. 10 instantiations therefore lead to 10 kiB code. 10 files each having 10 instantiations lead to 100 kiB machine code. Due to inline linkage, instantiations of the same specialization can get eliminated during linking, but nonetheless, the compiler *has* to generate the code.
>
> In practice, however, explicit instantiations are rare due to the resulting loss of flexibility. Therefore, templates are typically defined completely in header files. This is the primary reason why C++ programs tend to have very long compilation times.

[Slide 256] Out-of-Line Definitions

- Slightly weird syntax; sometimes preferable for interface readability

```
template <class T>
struct A {
    T value;
    A(T value);

    template <class R>
    R convert();
};

template <class T>
A<T>::A(T value) : value(value) { }

template <class T>
template <class R>
R A<T>::convert() { return static_cast<R>(value); }
```

> Some projects prefer out-of-line definitions in the header file for templates, because the declaration is more compact and therefore the interface becomes more readable.

[Slide 257] Templates: Example (1)

Quiz: What is the output of the program?

```
#include <print>
template <class T> class Foo {
  T value;
public:
  static unsigned count;
  Foo() { count++; }
};
template <class T> unsigned Foo<T>::count = 0;
int main() {
  Foo<int> foo1; Foo<long> foo2; Foo<Foo<int>> foo3;
  std::println("{}/{}", Foo<int>::count, Foo<long>::count);
  return 0;
}
```

A. (compile error) B. 1/1 C. 2/1 D. 3/3 E. 4/4

[Slide 258] Templates: Example (2)

Quiz: What is problematic about this code?

```
#include <array>
#include <print>
template <class T> struct ContainerContainer {
  T t;
  ContainerContainer() {}
  size_t size() const { return t.size(); }
};
int main() {
  ContainerContainer<std::array<int, 10>> cc1;
```

```
  ContainerContainer<long> cc2;
  return cc1.size();
}
```

A. Compile error: `const std::array<...>` has no method `size()`.
B. Compile error: `std::array` is only declared, but not defined.
C. Compile error: `long` has no method `size()`.
D. There is no problem, the program exits with status 10.

[Slide 259] Templates: Example (3)

Quiz: What is the output of the program?

```
#include <print>
#include <utility>
struct A { int a; };
int r(A&) { return 1; }
int r(A&&) { return 2; }

template <class T = void> int foo(T&& t) { return r(t); }
int main() {
  A a{12};
  std::println("{}/{}", foo(A{12}), foo(a));
  return 0;
}
```

A. (compile error) B. 1/1 C. 1/2 D. 2/1 E. 2/2

[Slide 260] Reference Collapsing[2]

- Template types can form references to references
- But: references of references are not allowed
- `T&& &&` ⇒ `T&&`
- `T& &&`, `T& &`, `T&& &` ⇒ `T&`

```
template <class T> int foo(T&& t) { return r(t); }
int main() {
  A a{12};
  foo(A{12}); // T is A&& => argument type is A&&
  foo(a); // T is A& => argument type is A&
  return 0;
}
```

[Slide 261] Template Argument Deduction[3]

- All template arguments must be known for instantiation
- But: not all have to be specified for class/function templates
- Based on function/constructor arguments; might fail when ambiguous
- Highly complex set of rules

[2] https://en.cppreference.com/w/cpp/language/reference#Reference_collapsing
[3] https://en.cppreference.com/w/cpp/language/template_argument_deduction

```
template <class T> T max(const T& a, const T& b);
int main() {
  int a = 0; long b = 42;
  max(a, b); // ERROR: Ambiguous deduction of T
  max(a, a); // OK, T = int
  max<int>(a, b); // OK
  max<long>(a, b); // OK

  std::unique_ptr ptr = make_unique<int>(42); // OK, T = int
}
```

[Slide 262] Templates: Argument Deduction (1)

Quiz: What is the output of the program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> size_t f(T* x) { return sizeof(*x); }
int main() {
  int* x = nullptr;
  std::println("{}", f(x));
  return 0;
}
```

A. (compile error) B. 4 C. 8 D. (undefined behavior)

[Slide 263] Templates: Argument Deduction (2)

Quiz: What is the output of the program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> size_t f(T* x) { return sizeof(*x); }

int main() {
  std::println("{}", f(nullptr));
  return 0;
}
```

A. (compile error) B. 0 C. (some positive integer) D. (undefined behavior)

7.2. auto Type

[Slide 264] `auto` Type[4]

- `auto` placeholder: deduce variable type from initializer
- Can (should!) be accompanied by usual modifiers (e.g. `const`, `*`, `&`)
  - `auto` not deduced to reference type, might cause unwanted copies

```
#include <unordered_map>
int main() {
```

[4] https://en.cppreference.com/w/cpp/language/auto

```
    std::unordered_map<int, const char*> intToStringMap;

    std::unordered_map<int, const char*>::iterator it1 =
        intToStringMap.begin(); // noone wants to read this

    auto it2 = intToStringMap.begin(); // much better
}
```

[Slide 265] `auto` Type – Examples

```
const int** foo();
struct A {
  const A& foo() { return *this; }
};

void bar() {
  auto f1 = foo(); // BAD: auto is const int**
  const auto f2 = foo(); // BAD: auto is const int**, f2 is const
  auto** f3 = foo(); // BAD: auto is const int

  const auto** f4 = foo(); // GOOD: auto is int

  A a;
  auto a1 = a.foo(); // BAD: auto is const A, copy
  const auto& a2 = a.foo(); // GOOD: auto is A, no copy
}
```

7.3. Variadic Templates

[Slide 266] Parameter Packs[5]

- Parameter pack: accept zero or more template arguments
- Type: `class ... Args` / Non-type: `type ... Args`
- Function parameters: `Args ... args`
- Appears in function parameter list of a variadic function template
- Parameter pack expression: `pattern...`
- Expands comma-separated list of `pattern`, which contains a parameter pack
  - E.g., `&args...` expands to `&arg1, &arg2, &arg3`

[Slide 267] Parameter Packs – Example

- Implementation somewhat difficult to write
- Straightforward way: tail recursion

```
#include <print>
void printElements() { } // recursion end

template <typename Head, typename... Tail>
void printElements(const Head& head, const Tail&... tail) {
  std::print("{}", head);
```

[5] https://en.cppreference.com/w/cpp/language/parameter_pack

```
  if (sizeof...(tail) > 0) // number of elements in Tail
    std::print(",␣");
  printElements(tail...);
}

int main() {
  // Output: "1, 2, 3, 3.14, hello, 4"
  printElements(1, 2, 3.0, 3.14, "hello", 4);
}
```

[Slide 268] Fold Expressions[6]

- Reduce parameter pack over binary operator
- `( pack op ... )` becomes $E_1 \circ (... \circ (E_{n-1} \circ E_n))$
- `( ... op pack )` becomes $((E_1 \circ E_2) \circ ...) \circ E_n$
- `( pack op ... op init )` becomes $E_1 \circ (... \circ (E_{n-1} \circ (E_n \circ I)))$
- `( init op ... op pack )` becomes $(((I \circ E_1) \circ E_2) \circ ...) \circ E_n$

```
template <typename R, typename... Args>
R reduceSum(const Args&... args) {
  return (args + ...);
}
int main() {
  return reduceSum<int>(1, 2, 3, 4); // returns 10
}
```

> Fold expressions enable to express semantics rather concise, but sometimes also very hard to understand. Variadic templates are generally used rather rarely.

7.4. Dependent Names

[Slide 269] Dependent Names (1)[7]

- In class template: class name and members refer to current instantiation

```
template <class T>
struct A {
  struct B { };

  B* b; // B refers to A<T>::B

  A(const A& other); // A refers to A<T>

  void foo();
  void bar() {
    foo(); // foo refers to A<T>::foo
  }
};
```

[6] https://en.cppreference.com/w/cpp/language/fold
[7] https://en.cppreference.com/w/cpp/language/dependent_name

[Slide 270] Dependent Names (2)

- Names dependent on template parameter types that are not members of the current instantiation are *not* considered as types by default

⇒ Needs `typename` disambiguator

- Can be omitted in some contexts, see reference

```
struct A {
  using MemberTypeAlias = float;
};
template <class T> struct B {
  using AnotherAlias = T::MemberTypeAlias; // no disambiguator required
  typename T::MemberTypeAlias* ptr; // disambiguator required
};
int main() {
  // outside template declaration => no disambiguator required
  B<A>::AnotherAlias value = 42.0f;
}
```

[Slide 271] Dependent Names (3)

- Similar rules apply for template names inside template definitions
- Name that is not member of current instantiation *not* considered as template

⇒ Needs `template` disambiguator

```
template <class T> struct A {
  template <class R>
  R convert(T value) { return static_cast<R>(value); }
};

template <class T> T foo() {
  A<int> a;
  return a.template convert<T>(42);
}
```

[Slide 272] Dependent Names: Motivation

Quiz: What is the output of the following program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> int fn(bool param, int a) {
  if (param) {
    T::member* a; return sizeof(a);
  }
  return 0;
}
struct S1 { static const int member = 5; };
struct S2 { using member = int; };
int main() {
  std::println("{}/{}", fn<S1>(true, 1), fn<S2>(true, 1));
}
```

A. (compile error)	B. 4/1	C. 4/4	D. 4/8	E. 8/8

> The reason why a `typename` disambiguator is sometimes required is syntactic ambiguity — without information whether a name is a type or a variable, C++ code is not parseable. Outside of templates, this information is known, but inside templates, it depends on the template parameter. To allow for parsing the template without this information, types must be explicitly disambiguated.

7.5. Explicit Specialization

[Slide 273] Explicit Specialization

- Sometimes, we want a different behavior for specific template arguments
- Example: different algorithm for a templated `find` function, e.g., binary search for array, linear search for linked
- Example: optimized storage for `bool`
- Explicit specialization: provide specific implementation for certain arguments
- Full specialization: all arguments are specified
- Partial specialization: some arguments are specified

[Slide 274] Full Specialization[8]

- `template <>` *declaration*
- Must come after the original template declaration

```
template <class T> class MyContainer {
    /* generic implementation */
};

template <> class MyContainer<bool> {
    /* specific implementation */
};

int main() {
    MyContainer<float> a; // uses generic implementation
    MyContainer<bool> b; // uses specific implementation
}
```

[Slide 275] Partial Specialization[9]

- `template <parameter-list> class` *name* `<`*argument-list*`>`
- Must come after the original template declaration
- Only class templates can be partially specialized

[8] https://en.cppreference.com/w/cpp/language/template_specialization
[9] https://en.cppreference.com/w/cpp/language/partial_specialization

```
template <class C, class T> class SearchAlgorithm {
    void find (const C& container, const T& value) {
        /* do linear search */
    }
};

template <class T> class SearchAlgorithm<std::vector<T>, T> {
    void find (const std::vector<T>& container, const T& value) {
        /* do binary search */
    }
};
```

[Slide 276] Specializations: Example

Quiz: What is the output of the following program?

```
#include <print>
template <class T> int foo(T) { return 1; }
template <> int foo<int*>(int*) { return 2; }

int bar(const int& l) { return foo(&l); }

int main() {
  std::println("{}", bar(10));
}
```

A. (compile error) B. 1 C. 2 D. (undefined behavior)

[Slide 277] Traits

- Trait classes: provide generic way to access information about types

```
template <class> class GraphTraits {};

template <class GraphT>
std::vector<typename GraphTraits<GraphT>::NodePtr>
    traverseGraph(const GraphT& graph) {
  using NodePtr = GraphTraits<GraphT>::NodePtr;
  NodePtr start = GraphTraits<GraphT>::getFirstNode(graph);
  // ...
}

template <> // Specialization of traits class for specific graph
class GraphTraits<MyGraph> {
  using NodePtr = MyNode*;
  NodePtr getFirstNode(const MyGraph& graph) { /* ... */ }
  // ...
}
```

> While template alone already allow for a type-generic implementation, this is often not sufficient when actually *doing* something with the type. Constraints (see later) already allow to express certain requirements, but this is in general not flexible enough — for example, when using a data structure from a library which follows a different naming

convention. In such cases, using traits classes allows to use such data structures as-is, while having a unified interface for the required functionality.

7.6. Type Traits

[Slide 278] Type Traits (1)[10]

- Specialization useful for querying information about types themselves (traits)

```
// NB: use std::is_same instead
// Base case
template <class T1, class T2>
struct is_same { static constexpr bool value = false; };

// Specialization for case where both types are the same
template <class T>
struct is_same<T, T> { static constexpr bool value = true; };

#include <cstdint>
static_assert(is_same<int, long>::value == false);
static_assert(is_same<int, const int>::value == true);
static_assert(is_same<int, int32_t>::value == true);
```

[Slide 279] Type Traits (2)

```
// NB: use std::remove_reference instead
template<class T> struct remove_reference { using type = T; };
template<class T> struct remove_reference<T&> { using type = T; };
template<class T> struct remove_reference<T&&> { using type = T; };

template<class T>
using remove_reference_t = typename remove_reference<T>::type;

static_assert(std::is_same<int, remove_reference_t<int>>::value);
static_assert(std::is_same<int, remove_reference_t<int&>>::value);
static_assert(std::is_same<int, remove_reference_t<int&&>>::value);
static_assert(std::is_same<int*, remove_reference_t<int*>>::value);
```

[Slide 280] Implementation of `std::move`

- With type traits, `std::move` becomes easily implementable

```
template <class T>
std::remove_reference_t<T>&& move(T&& t) {
  return static_cast<std::remove_reference_t<T>&&>(t);
}
```

The `remove_reference` is required due to reference collapsing; the function should accept an lvalue or rvalue reference but always return an rvalue reference.

This is just the conceptual implementation. Standard library code tends to be a lot

[10] https://en.cppreference.com/w/cpp/header/type_traits

> less readable — all non-fixed names must begin with an underscore (such names are reserved by the standard; any other name could be collide with a user-defined name or macro) and often makes heave use of macros to adjust for different C++ standard versions and compilers.

7.7. Constraints

[Slide 281] Type Constraints[11]

- Template might assume certain things about parameters
  - E.g., member function, copyable, moveable
  - Effectively "duck typing"
- Might lead to (horrible) compilation errors when used with incorrect arguments
- Constraint: requirements on template arguments
- Concept: named set of requirements

[Slide 282] Type Constraints – `requires` Clause[12]

- `requires <`*`constant-expression`*`>` – apply constraint to template

```
#include <concepts>

template <class T> requires true // useless, but valid
void fn() {}

template <class T> requires std::floating_point<T>
T fdiv1(T a, T b) {
  return a / b;
}

// template <Concept X> == template <class X> requires Concept<X>
template <std::floating_point T>
T fdiv2(T a, T b) {
  return a / b;
}
```

[Slide 283] Type Constraints and Concepts

- Can also be combined with `&&` and `||`
- Greatly improve safety: fewer implicit assumptions
- Improve quality of error messages
- We will revisit constraints and concepts later in the lecture

[Slide 284] Templates – Summary

- Templates enable compile-time specialization of classes/functions/types
- Instantiation either explicit or implicit on use

[11] https://en.cppreference.com/w/cpp/language/constraints
[12] https://en.cppreference.com/w/cpp/language/constraints#Requires_clauses

  - Implicit instantiation typically used in practice
- Template arguments sometimes can be deduced from (constructor) arguments
- Names dependent on parameter types may need type disambiguator
- Template specialization: different implementation for specific arguments
- Trait classes use specialization to provide generic interface
- Constraints can express requirements on template arguments
- `auto` exposes type deduction to variable declarations

[Slide 285] Templates – Questions

- When is a template instantiated?
- What is the benefit of explicit instantiation?
- Where to template definitions go?
- How to provide an out-of-line definition for a method in a template class?
- When is a `typename` disambiguator required?
- Why is using `auto` without modifiers sometime problematic?
- What are use cases of template specialization?
- How to express requirements on type template arguments?

8. Containers and Iterators

8.1. Utilities

[Slide 287] `std::optional`[1]

- `std::optional<T>` (`<optional>`): value that might not exist
- Can be empty (no value) or non-empty (holding a value)
- Implicit conversion to `bool`, access contained value with `*` or `->`

```
std::optional<std::string> mightFail(unsigned arg) {
  if (arg < 7) {
    return "lt␣7"; // equiv to: std::optional<std::string>("lt 7")
  } else {
    return std::nullopt; // alternatively: return {};
  }
}
void foo(unsigned n) {
  if (auto optStr = mightFail(n))
    std::println("{}", optStr->size()); // prints: 4
}
```

[Slide 288] Optional Reference

Quiz: What is the most efficient way to return an optional reference?

A. `std::optional<Foo&>`
B. `std::optional<Foo*>`
C. `std::optional<std::reference_wrapper<Foo>>`
D. `Foo*`

[Slide 289] `std::pair`[2]

- `std::pair<T, U>` (`<utility>`): pair of two values
- Members can be accessed with `first` and `second`
- Constructible with constructor or `std::make_pair`

```
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<int, double>
p1 < p2; // true
```

[1] https://en.cppreference.com/w/cpp/utility/optional
[2] https://en.cppreference.com/w/cpp/utility/pair

[Slide 290] `std::tuple`[3]

- `std::tuple<...>` (`<utility>`): tuple of n values
- Members can be accessed with `std::get<i>()`
- Constructible with constructor or `std::make_tuple`

```
std::tuple<int, double, char> t1(123, 4.56, 'x');
std::get<1>(t1); // == 4.56
auto p2 = std::make_tuple(456, 1.23, 'y');
// p2 has type std::tuple<int, double, char>
p1 < p2; // true
```

[Slide 291] Structured Bindings[4]

- `auto [a, b] = t;` initialized with `std::get<0>(t)` and `std::get<1>(t)`
- Also with `auto&` and `const auto&` for references to elements

```
auto t = std::make_tuple(1, 2, 3);
auto [a, b, c] = t; // a, b, c have type int
auto p = std::make_pair(4, 5);
auto& [x, y] = p; // x, y have type int&
x = 123; // p.first is now 123
```

[Slide 292] Using Pair/Tuple

- Pair/tuple convey no information about semantics
- User-defined types often preferable, esp. in public interfaces

$\Rightarrow$ Use `std::pair`/`std::tuple` sparingly

```
struct Rational {
  long numerator;
  long denominator;
};
std::pair<long, long> canonicalize(long, long); // BAD
Rational canonicalize(const Rational&); // BETTER
```

[Slide 293] `std::variant`[5]

- Type which holds exactly one of the alternative types
- Type-safe, alternative share same underlying storage $\rightsquigarrow$ smaller size
- Accessible with `std::get`, `std::holds_alternative`

```
std::variant<int, double> v; // holds either an int or a double

v = 42; // now holds an int
assert(std::holds_alternative<int>(v));
assert(std::get<int>(v) == 42);

v = 1.0; // now holds a double
// get_if returns pointer to active value, or nullptr
```

[3] https://en.cppreference.com/w/cpp/utility/tuple
[4] https://en.cppreference.com/w/cpp/language/structured_binding
[5] https://en.cppreference.com/w/cpp/utility/variant

```
assert(*std::get_if<double>(&v) == 1.0);
assert(std::get_if<int>(&v) == nullptr);
```

8.2. Iterators

[Slide 294] Iterators[6]

- Standard library provides various containers, code might define custom ones
- Problem: different containers can have different access methods

⇝ containers not easily exchangable

- Solution: abstract over element access with iterators
  - Same pointer-like interface for all containers

⇒ Allows for easy exchange of container type
  - Very useful in templates specialized on containers
- Containers define:
  - `begin()` – iterator pointing to first element
  - `end()` – iterator pointing to the first element *after* the container

> `begin()` returns an iterator to the first element; if the container is empty, it returns the same as `end()`. `end()` returns an iterator *past* all elements (similar to a past-the-end pointer of an array). `end()` is therefore not dereferenceable.

[Slide 295] Iterators: Usage Example

```
#include <array>
#include <print>
int main() {
  std::array<int, 2> arr{1, 2};
  auto it = arr.begin();
  assert(*it == 1);
  ++it; // prefer pre-increment
  assert(*it == 2);
  ++it;
  assert(it == arr.end()); // end iterator not dereferencable (UB)

  for (auto it = arr.begin(); end = arr.end(); it != end; ++it)
    std::println("{}", *it);
}
```

> Prefer pre-increment — post-increment creates a copy of the iterator.

[Slide 296] Range-Based `for` Loop[7]

- for-range loop is syntactic sugar for:

[6] https://en.cppreference.com/w/cpp/iterator
[7] https://en.cppreference.com/w/cpp/language/range-for

  - Calling `begin()` and `end()` of the range
  - Looping until the iterator equals the end iterator
  - Defining variables inside the loop body from the iterator

```
#include <array>
#include <print>
int main() {
  std::array<int, 2> arr{1, 2};
  for (int& x : arr)
    x += 5;
  // ... is identical to:
  for (auto it = arr.begin(); end = arr.end(); it != end; ++it) {
    int& x = *it;
    x += 5;
  }
}
```

> Prefer using the for-range loop for iterating over containers, as it is more readable. However, it is not usable if the iterator is required, e.g. for removing elements from the container.

[Slide 297] Input/Output Iterator

- Concepts: `std::input_iterator`/`std::output_iterator`
- Required features:
  - `it1 == it2` – whether iterators point to the same position
  - `*it`, `it->` – dereferencing
  - `++it`, `it++` – incrementing
  - Input iterator: dereferenced iterator can only be read
  - Output iterator: dereferenced iterator can only be written to
- Single-pass only: not decrementable, two iterators might yield different values

[Slide 298] Forward/Bidirectional Iterator

- Concepts: `std::forward_iterator`/`std::bidirectional_iterator`
- Forward iterator – required features:
  - All features shared by input/output iterator
  - Multi-pass guarantee: `it1 == it2` implies `++it1 == ++it2`
- Bidirectional iterator – forward iterator with:
  - `--it`, `it--` – decrementing (walking backwards)

[Slide 299] Random Access/Contiguous Iterator

- Concepts: `std::random_access_iterator`/`std::contiguous_iterator`
- Random access iterators – bidirectional iterator with:
  - `it[]` – random access
  - Relational operators, e.g. `it1 < it2`

  - Incrementable/decrementable by any amount, e.g. `it + 2`, `it -= 5`
- Contiguous iterator – random access iterator with:
  - Elements are stored contiguously in memory
  - `&*(it + n)` equivalent to `(&*it) + n`

[Slide 300] Implementing Iterators for a Linked List

We will write our custom singly-linked list in C++. We use `std::unique_ptr` to handle memory management. We start with the basic implementation:

```
#include <memory>
#include <utility>
#include <iterator>

template <class T> class ForwardList {
public:
  using reference = T&;
  using const_reference = const T&;
  using pointer = T*;
  using const_pointer = const T*;

private:
  struct Element {
    std::unique_ptr<Element> next;
    T value;

    // We need two constructors, one for initializing with a copy and
    // one for initializing with a move.
    Element(std::unique_ptr<Element> next, const T& t)
        : next(std::move(next)), value(t) {}
    Element(std::unique_ptr<Element> next, T&& t)
        : next(std::move(next)), value(std::move(t)) {}

    // we could explicitly delete copy/move constructor/assignment -- they
    // are not needed. Copying isn't possible anyway due to unique_ptr
  };
  std::unique_ptr<Element> first;

public:
  ForwardList() = default;
  // no custom copy/move constructor/assignment needed -- rule of zero

  // Some simple accessors
  bool empty() const { return !first; }
  reference front() { return first->value; }
  const_reference front() const { return first->value; }

  // Prepending elements is fast
  void push_front(const T& t) {
    first = std::make_unique<Element>(std::move(first), t);
  }
  void push_front(T&& t) {
    first = std::make_unique<Element>(std::move(first), std::move(t));
```

```
  }
  // Deleting the first element is also fast
  void pop_front() {
    first = std::move(first->next);
  }
  // Remove all elements, will recursively delete all elements
  void clear() { first = nullptr; }

  // Now we want to allow iterating over the elements of the list.
  // We need to implement begin() and end().
  // But we need our own iterator type for this.
  // We have a separate const iterator which prohibits modifications
  // (and is slightly easier to implement) -- so we'll do this first.
  class const_iterator {
    const Element* elem;
  public:
    // Required member types for iterator
    using difference_type = std::ptrdiff_t;
    using value_type = T;

    // Iterators should be default-constructible
    const_iterator(const Element* elem = nullptr) : elem(elem) {}

    // Straightforward implementation of required operators
    const_reference operator*() const { return elem->value; }
    const_pointer operator->() const { return &elem->value; }
    const_iterator& operator++() { elem = elem->next.get(); return *this; }
    // Post-increment makes copy, calls pre-increment, and returns the old copy
    const_iterator operator++(int) { auto tmp = *this; ++*this; return tmp; }
    bool operator==(const const_iterator& other) const { return elem == other.elem; }
  };

  // Let the compiler check that we have implemented all required methods.
  static_assert(std::forward_iterator<const_iterator>);

  // begin() and end() are simple; for an empty list, begin() == end().
  const_iterator begin() const { return const_iterator{first.get()}; }
  const_iterator end() const { return const_iterator{}; }

  // ...
};
```

At this point, we can also write some tests for our implementation:

```
int main() {
  ForwardList<int> l;
  l.push_front(1);
  l.push_front(2);
  l.push_front(3);
  auto it = l.begin();
  assert(it == l.begin());
  assert(*it == 3);
  ++it;
  assert(it == ++l.begin());
  assert(*it == 2);
  ++it;
```

```
  assert(*it == 1);
  ++it;
  assert(it == l.end());
  for (const int& e : l)
    std::println("{}", e);
  return 0;
}
```

[Slide 301] Insertion and Removal

- Containers generally use iterators for removing elements
  - Already have some handle to the element $\rightsquigarrow$ use it
  - Especially important for data structures with non-$\mathcal{O}(1)$ access
  - Typically: `erase(iterator)`
- Likewise: insertion at a specific point
- **Important: might invalidate the used or some/all other iterators!**

In our linked list, we want out non-`const` iterator not just to expose a mutable reference to the stored objects, but also to allow removing them. This is not trivial for a singly-linked list.

How to remove elements from a singly-linked list?

No back pointers – how to update previous next pointer?

There are multiple ways to implement this. One could store a pointer to the previous element in the iterator. Or, as we will do it, store a pointer to the previous next pointer (beginning: pointer to the first pointer) only.

```
class ForwardList {
  // ...

  class iterator {
    friend class ForwardList; // for erase etc.

    // NB: pointer to unique_ptr. Pointer to the unique_ptr of the current element.
    std::unique_ptr<Element>* elem;
    // Access actual element, as the pointer-to-pointer might be null.
    Element* element() const { return elem ? elem->get() : nullptr; }

  public:
    using difference_type = std::ptrdiff_t;
    using value_type = T;

    // Iterator should be default-constructible
    iterator(std::unique_ptr<Element>* elem = nullptr) : elem(elem) {}

    // Nothing special here.
    reference operator*() const { return element()->value; }
```

```
    pointer operator->() const { return &element()->value; }
    iterator& operator++() { elem = &element()->next; return *this; }
    iterator operator++(int) { auto tmp = *this; ++*this; return tmp; }
    bool operator==(const iterator& other) const { return element() == other.element(); }
  };

  static_assert(std::forward_iterator<iterator>);

  // non-const begin() and end()
  iterator begin() { return iterator{&first}; }
  iterator end() { return iterator{}; }

  // Erase invalidates it (we remove the element). Return the iterator to the
  // next element.
  iterator erase(iterator it) {
    // Move next element into previous next pointer or first pointer.
    // This overwrites a unique_ptr, causing the element to get deleted.
    *it.elem = std::move((*it.elem)->next);
    return it;
  }
};
```

We can also test this implementation:

```
int main() {
  ForwardList<int> l;
  l.push_front(1);
  l.push_front(2);
  l.push_front(3);
  l.erase(l.begin());
  auto it = l.begin();
  assert(*it == 2);
  ++it;
  assert(*it == 1);
  it = l.erase(it); // erase 1
  assert(it == l.end());

  for (int& e : l) {
    e += 10; // we can modify the elements inside the list
    std::println("{}", e);
  }
  // output: 12
}
```

The use of overloaded operators for iterators has one major advantage when iterating over contiguous allocations: the iterator can be defined to be just a `T*`. (Note how the pointer type fulfills all requirements.)

8.3. Vector and Span

[Slide 302] Containers in Standard Library: Overview

- Container: object that stores collection of other objects
- Types of elements specified as template parameter(s)

- Sequential: optimized for sequential access
  - E.g., `std::array`, `std::vector`, `std::list`
- Associative: sorted, optimized for search ($\mathcal{O}(\log n)$)
  - E.g., `std::set`, `std::map`
- Unordered associative: hashed, optimized for search ($\mathcal{O}(n)$, amortized $\mathcal{O}(1)$)
  - E.g., `std::unorderd_set`, `std::unorderd_map`

[Slide 303] `std::vector`[8]

- Array that can dynamically grow size
- Elements stored contiguously in memory, access via `data()`
- Preallocates memory for a certain amount of elements (*capacity*)
  - Default: exponential growth; can `reserve()` to reduce reallocations
- Random access: $\mathcal{O}(1)$
- Insert/remove at end: $\mathcal{O}(1)$ (amortized)
- Insert/remove at other position: $\mathcal{O}(n)$

[Slide 304] `std::vector` Example

```
std::vector<int> fib = {1,1,2,3};
assert(fib[1] == 1);
int* fib_ptr = fib.data();
assert(fib_ptr[2] == 2);
fib[3] = 43;
fib.data()[1] = 41; // fib is now 1, 41, 2, 43

fib.push_back(5); // fib is now 1, 41, 2, 43, 5
assert(fib.size() == 5);
assert(fib.back() == 5);
fib.pop_back(); // fib is now 1, 41, 2, 43
auto it = fib.begin(); it += 2;
fib.insert(it, 99); // fib is now 1, 41, 99, 2, 43
it = fib.begin() + 2;
fib.erase(it); // fib is now 1, 41, 2, 43

fib.clear(); // remove all elements
assert(fib.empty());
```

[Slide 305] `std::vector` Example

Quiz: What is problematic about this code?

```
#include <vector>
void func(std::vector<int>& v) {
  for (const int& i : v)
    if (i > 1)
```

[8] https://en.cppreference.com/w/cpp/container/vector

```
        v.insert(v.begin(), -i);
}
```

A. Compile error: Cannot get `const` reference for element.
B. Compile error: `insert()` needs an index as first parameter.
C. Undefined behavior: after the `if` body, an invalidated iterator is used.
D. There is no problem.

[Slide 306] `std::vector` Example

Quiz: How could this code be improved?

```
#include <array>
#include <cstddef>
#include <vector>
template <size_t N> void func(std::vector<std::array<int, N>>& v, int x) {
  std::array<int, N> a;
  for (size_t i = 1; i < N; i++) a[i] = a[i-1] * x + i;
  v.push_back(a);
}
```

A. Instead of copying the array, use `std::move` in `push_back`.
B. Construct the array in-place in the vector, then modify that.
C. Make `a` a reference to reduce stack memory usage.
D. There is nothing to improve.

[Slide 307] `std::vector`: Emplacing Elements

- `emplace(_back)`: construct element in place to avoid copying/moving
- Arguments forwarded to constructor, returns reference to object

```
struct ExpensiveToCopy { /* ... */ };

std::vector<ExpensiveToCopy> v;
ExpensiveToCopy e1;
e1.foo();
v.push_back(e1); // BAD: copy
v.push_back(std::move(e1)); // Better, but might still be expensive

// Best: element constructed in its final place in the vector
ExpensiveToCopy& e2 = v.emplace_back();
e2.foo();
```

[Slide 308] `std::vector`: Reserving Memory

- `reserve`: size hint to avoid reallocations
- `capacity`: get currently allocated size

```
std::vector<int> v;

v.reserve(1'000'000); // allocate memory for 1M elements
assert(v.capacity() == 1'000'000);
```

```
assert(v.size() == 0); // the vector is still empty!

for (int i = 0; i < 1'000'000; ++i) {
  vec.push_back(i); // no reallocations in this loop
}
```

[Slide 309] `std::vector` Reserve Example

Quiz: What is problematic about this code?

```
std::vector<int> func(unsigned n) {
  std::vector<int> res;
  res.reserve(n);
  std::vector<int>::iterator it = res.end();
  for (size_t i = 0; i < n; i++) {
    res.push_back(i * i);
    if (i % 3 == 0) it = res.begin() + i;
  }
  res.push_back(*it);
  return res;
}
```

A. Returning a `vector` by value is very expensive.
B. The last `push_back` causes an out-of-bounds write.
C. `it` is invalidated immediately in the next loop iteration.
D. There is no problem.

[Slide 310] `std::span`[9]

- Reference to contiguous array of objects; pair of pointer/length
- Supports iteration, subscript, `size()`, `data()`
- `subspan()`: sub-region, no elements copied

```
void printValues(std::span<const int> is) {
  for (auto i : is) std::print("{} ", i);
}
std::vector<int> values{1, 2, 3, 4};
std::span<int> valuesRef = values;
valuesRef[2] = 4;
printValues(values); // prints "1 2 4 4 "
```

- Prefer `std::span` over reference to `std::array`, `std::vector`, ...
- Pass `std::span` by value (it is already a reference)
- Prefer `std::span<const T>` if possible

> As `std::span` is basically a reference to an array, pass it by value. Taking a `std::span` as parameter is preferable over a reference to a vector (if the number of elements is not modified), as it increases the flexibility of the function.

[9] https://en.cppreference.com/w/cpp/container/span

[Slide 311] `std::span` Example

Quiz: What is problematic about this code?

```
void func(std::span<const int> cs, std::vector<int>& v) {
  for (int c : cs)
    if (c < 0)
      v.push_back(c);
}
int main() {
  std::vector<int> v{-1, 10, -100, 20};
  func(v, v);
}
```

A. Compile error: Must be `const int c : cs`
B. Passing a vector as `span` precludes passing it as reference at the same time.
C. The `push_back` invalidates the iterator of the loop.
D. There is no problem.

8.4. Map and Set

[Slide 312] `std::unordered_map`[10]

- `std::unordered_map<KeyT, ValueT>` (`unordered_map`)
  - Accepts custom hash and comparison functions as extra template parameters
- Container that stores key–value pairs with unique key
- Internally a hash table, amortized $\mathcal{O}(1)$ search/insert/remove

```
std::unordered_map<unsigned, double> grades;
grades[12340001] = 1.3;
grades.insert({12340042, 2.7});
grades.emplace(12340123, 5.0); // emplace = construct in-place
assert(grades[12340042] == 2.7);

auto it = grades.find(12340001); // search
if (it != grades.end()) { // found
  assert(it->first == 12340001); // key
  assert(it->second == 1.3); // value
}
assert(grades.contains(12340001));
```

[Slide 313] Unordered Map: Misleading Usage

Quiz: Which answer is NOT correct?

```
std::optional<double> lookup(std::unordered_map<unsigned, double>& map,
    unsigned key) {
  if (map[key])
    return map[key];
```

[10] https://en.cppreference.com/w/cpp/container/unordered_map

```
    return -1.0;
}
```

A. `key` is always inserted into the map.
B. If the stored value is zero, `-1` is returned.
C. `map` is not modified and therefore should be a const reference.
D. The map is searched twice, which is avoidable and inefficient.

[Slide 314] Unordered Map: Modification

Insertion:

- `operator[]` – get reference to value, insert and default-construct if missing
- `insert` – insert if missing and copy/move construct
  - Returns `std::pair<iterator,bool>`; `second` true iff insertion happened
- `emplace` – construct in-place if missing
- Iterator invalidation: only on rehash

Removal:

- `erase(iterator)`/`erase(key)` – remove element
  - Iterator invalidation: only iterator for `key`
- `clear` – remove all elements
  - Iterator invalidation: all

[Slide 315] `std::map`[11]

- `std::map<KeyT, ValueT>` (`<map>`) – map sorted by keys
- Interface largely similar to `unordered_map`
  - Also supported `upper_bound()`/`lower_bound()` – return iterator to first greater/not lower element
- Internally a tree (typically R/B-tree), $\mathcal{O}(\log n)$ search/insert/remove
- **Only use of *sorted* keys are required!**

[Slide 316] `std::unordered_set` and `std::set`

- `std::unordered_set<KeyT>` (`<unordered_set>`) – hash set
- `std::set<KeyT>` (`<set>`) – set sorted by keys
- Largely similar to maps without values
  - Similar internal representation, methods, complexities
- Keys must not be modified

[11] `https://en.cppreference.com/w/cpp/container/map`

8.5. String

[Slide 317] `std::string`[12]

- `std::string` (`<string>`) (alias for `std::basic_string<char>`)
- Class for (mutable) character sequences
- Manages memory and knows its length (unlike C strings)
- Access to underlying C-string: `c_str()`
- Prefer `std::string` over C-style strings (`char*`)!

```
std::string s; // default-constructs, empty string
assert(s.size() == 0);

std::string s_constructed("my␣literal");
std::string s_assigned = "hi";
s_assigned[0] = 'H';
std::println("{}␣{}", s_assigned, s_assigned[1]); // prints: "Hi i"
```

[Slide 318] `std::string`: Null Bytes

Quiz: What is the output of the following program?

```
#include <print>
#include <string>
int main() {
  std::string s1 = "null\0byte";
  std::string s2("null\0byte", 9);
  std::println("{}/{}", s1, s2);
  return 0;
}
```

A. Compile error: String literals cannot include null-bytes
B. Undefined behavior: `std::string` cannot include null-bytes
C. $\texttt{null}_0\texttt{byte/null}_0\texttt{byte}$
D. $\texttt{null/null}_0\texttt{byte}$
E. `null/null`

[Slide 319] `std::string`: Operations

- `==`, `<=>`: lexicographical comparison of full strings
- `size()`: number of characters in string
- `empty()`: whether string is empty
- `find()`: offset of first occurrence of substring, or `std::string::npos`
- `append()`, `+=`: append string/char, might cause memory allocation
- `+`: concatenate into new heap-allocated string
- `substr()`: new `std::string` containing substring
  - This is often *not* what you want!

[12] `https://en.cppreference.com/w/cpp/header/string`

> Be careful about string and especially constructing new strings (e.g., by passing them by value, `substr`, `operator+`): these create a new copy of the string, which often means a new heap allocation, which is expensive. Therefore, use string views where possible.

[Slide 320] `std::string_view`[13]

- Read-only view on existing string
- Similar to `span<const char>`: just a pointer and a length

⇝ Creation, substring, copying is constant time (linear for `std::string`)

- **Prefer `std::string_view` over `std::string`/`std::string&`**

```
std::string s = "garbage␣garbage␣garbage␣interesting␣garbage";
std::string sub = s.substr(24,11); // With string: O(n)
// With string view:
std::string_view s_view = s; // O(1)
std::string_view sub_view = s_view.substr(24,11); // O(1)

bool is_eq_naive(std::string a, std::string b) {return a == b; }
bool is_eq_views(std::string_view a, std::string_view b) { return a == b; }
is_eq_naive("abc", "def"); // 2 allocations at runtime
is_eq_with_views("abc", "def"); // no allocation at runtime
```

[Slide 321] `std::string`: Implementation

- Different standard libraries have different implementations[14]
- Typically: pointer, size, capacity
  - Pointer (can) to heap memory, deleted on destruction
- Typically: small-buffer optimization
  - Most strings are small, heap allocations are expensive

  ⇝ Store small buffer (e.g., 15 bytes) inline in `std::string`
  - Downside: more operations invalidate iterators
  - Permitted by C++ standard

[Slide 322] Small Buffer Optimization

Quiz: Why does `std::vector` not implement small-buffer optimization?

A. Not very useful ⇒ no one implemented it so far.
B. Insertion would no longer be amortized $\mathcal{O}(1)$.
C. Reduce memory usage by not having inline space.
D. Moving a vector must not invalidate iterators.

[13] https://en.cppreference.com/w/cpp/string/basic_string_view
[14] https://devblogs.microsoft.com/oldnewthing/20240510-00/?p=109742

[Slide 323] Containers and Iterators – Summary

- Standard library provides several utility and container templates
- Simple pairs/tuples; can be extracted with structured bindings
- Iterators provide unified pointer-like interface for container element access
- Modifications of containers typically invalidate iterators
- Vector: dynamically sized array, most popular container
- (Unordered) map/set: associative containers
  - Ordered containers typically less efficient
- String: character sequence with managed storage
- String view/span: view into array or string
- Containers good enough to not *immediately* write a custom implementation

[Slide 324] Containers and Iterators – Questions

- When do iterators get invalidated? How does this vary for different containers and their operations?
- Why does iterator invalidation frequently cause problems in practice?
- How does a range-based for loop work?
- Why is are unordered maps/sets preferable over ordered maps/sets?
- What are the benefits of `std::string` over C-style strings?
- When to use `std::span`/`std::string_view` and pass them as parameters?
- Why is small-buffer optimization often beneficial/wanted?

9. Algorithms, Functions, and Lambdas

9.1. Function Objects

[Slide 326] Function Objects[1]

- Functions are no objects
  - Cannot be passed as parameters, no state, etc.
- *FunctionObject* named requirement for some type `T`:
  - `T` must be an object
  - For an instance `f` of `T`: `f(args)` must be defined
- Also referred to as *functors*

[Slide 327] Function Pointers[2]

- Functions are not objects, but have addresses
  - Location in memory where the code resides

⇝ Allows declaration of function pointers: `ret-ty (*identifier)(arg-tys)`

- Function pointers satisfy requirements of *FunctionObject*

```
int add(int p1, int p2) { return p1 + p2; }
int callFn(int (*fn)(int, int), int p1, int p2) {
  return fn(p1, p2); // automatic dereference, equivalent to (*fn)(p1, p2)
}

int main() {
  // add gets implicitly converted to function pointer, equiv. to &add
  int res = callFn(add, 1, 1); // 2
}
```

> Prefer type aliases (e.g., `using BinaryFn = int(*)(int, int);`) when working with function pointer types. Program can quickly become hard to read otherwise.

[Slide 328] Function Pointers

Quiz: What is the output of the program?

```
#include <print>
int fn(int p, int q = 1) { return p * q; }
int callFn(int (*fn)(int), int p) {
```

[1] https://en.cppreference.com/w/cpp/named_req/FunctionObject
[2] https://en.cppreference.com/w/cpp/language/pointer

```
    return (*fn)(p);
  }
  int main() {
    std::println("{}", callFn(fn, 1));
  }
```

A. Compile error: `fn` has type `int(int, int)`
B. Compile error: cannot dereference function pointer
C. Undefined behavior: default arguments become undefined
D. Guaranteed output: `1`

[Slide 329] Member Function Pointers

- Non-static member functions take an implicit parameter, `this`

⇒ Special pointers to members of a class

```
struct S {
  int a;
  int x(int b, int c) { return a + b + c; }
};

int main() {
  int (S::* memberFnPtr)(int, int) = &S::x;

  S s{32};
  int r1 = (s.*memberFnPtr)(2, 8); // 42
  S* sp = &s;
  int r2 = (sp->*memberFnPtr)(2, 8); // 42
}
```

> Member function pointers are rarely needed. There are also member data pointers with a similar syntax, which are even more exotic and rarely useful.

[Slide 330] Stateful Function Objects

- So far: functions are stateless have to pass all state as parameters
- Function objects can be implemented as regular class

⇝ Keep arbitrary state as members

```
struct Adder {
  int v;
  int operator()(int param) const {
    return param + v;
  }
};
int main() {
  Adder add42{42};
  int a = add42(10); // 52
  add42.v = 10;
  int b = add42(0); // 10
}
```

[Slide 331] Stateful Function Objects

Quiz: What is the output of the program?

```
#include <print>
struct Accumulator {
  int v;
  int operator()(int param) {
    return v += param;
  }
};
int main() {
  Accumulator acc{10};
  int a = acc(10) + acc(20);
  int b = acc(1) + acc.v;
  std::println("{}/{}", a, b);
}
```

A. (compile error) B. (multiple outputs correct) C. `60/81` D. `60/82` E. `70/82`

[Slide 332] Lambda Expressions (1)[3]

- Function pointers so far have some limitations:
  - Cannot have "local" functions within other functions
  - Cannot capture environment – have to pass all state as parameters
- Lambda expressions: construct closure (function with environment)
- `[captures](params) -> ret-ty { body }`
  - Captures specify parts of environment that should be stored, can be empty
  - Lambda without captures are implicitly converted to function pointers
  - Return type can be omitted if deducible from return statements in body
- Lambda expressions have a unique, unnamed type

⇝ Rely on `auto`/argument deduction when assigning lambda

[Slide 333] Lambda Expressions

```
int callFn(int (*fn)(int, int), int p1, int p2) {
  return fn(p1, p2); // automatic dereference, equivalent to (*fn)(p1, p2)
}

int main() {
  auto lambda = [](int p1, int p2) {
    return p1 + p2;
  };

  // lambda gets implicitly converted to function pointer
  int res = callFn(add, 1, 1); // 2
  int foo = lambda(2, 4); // 6
}
```

[3] https://en.cppreference.com/w/cpp/language/lambda

[Slide 334] Lambda Expressions

- Lambda types are really unique:

```
// ERROR: Compilation will fail due to ambiguous return type
auto getFunction(bool first) {
  if (first) {
    return []() {
      return 42;
    };
  } else {
    return []() {
      return 42;
    };
  }
}
```

[Slide 335] Lambda Captures

- Capture: specify what constitutes the state of a lambda expression
- Refer to automatic variables/`this` in surrounding scopes
- Captured variables can be used in lambda like regular variables
- Capture *by-copy*: `[var]` or `[var = initializer]`
  - Variable *copied* into lambda state on creation
- Capture *by-reference*: `[&var]` or `[&var = initializer]`
  - Variable *reference* stored lambda state on creation
- Each variable may be captured at most once

[Slide 336] Lambda Captures: Example

```
int main() {
  int i = 42;

  // Lambda stores a copy of i
  auto lambda1 = [i]() { return i + 42; };
  // Lambda stores a reference to i
  auto lambda2 = [&i]() { return i + 42; };

  i = 0;

  int a = lambda1(); // a = 84
  int b = lambda2(); // b = 42
}
```

[Slide 337] Lambda Capture: Default Captures

- First capture can be a capture-default: `=` (copy)/`&` (reference)
- Allows use of any variable in surrounding scope
- Diverging capture types can be specified for individual variables

```
int main() {
   int i = 0;
```

```
  int j = 42;

  auto lambda0 = [=](){}; // j and i by-copy
  auto lambda1 = [&](){}; // j and i by-reference
  auto lambda2 = [&, i](){}; // j by-reference, i by-copy
  auto lambda3 = [=, &i](){}; // j by-copy, i by-reference
  auto lambda4 = [&, &i](){}; // ERROR: non-diverging capture types
  auto lambda5 = [=, i](){}; // ERROR: non-diverging capture types
}
```

[Slide 338] Lambda Captures

Quiz: What is problematic about this code?

```
auto createAdder(int n) {
  return [&](int x) {
    return x + n;
  };
}
int main() {
  auto add42 = createAdder(42);
  return add42(10);
}
```

A. Compile error: return type must be named ⇒ cannot return lambda.
B. Compile error: cannot copy lambda.
C. Undefined behavior: function call uses a dangling reference.
D. No problem: the program exits with status 52.

[Slide 339] Lambda Captures

Quiz: What is the output of the program?

```
struct Too {
  int x;
  auto makeBuzzer() {
    return [*this](int y) { return x + y; };
  }
};
int main() {
  auto buzzer = Too{32}.makeBuzzer();
  std::println("{}", buzzer(10));
}
```

A. Compile error: cannot capture `*this`.
B. Compile error: `x` is not captured.
C. Undefined behavior: `x` used after `Too` got destroyed.
D. Guaranteed output: `42`

[Slide 340] Lambda Capture: `this`

- `*this` (copy)/`this` (reference)

```
struct Foo {
  int i = 0;

  void bar() {
    auto lambda1 = [*this]() { return i + 42; };
    auto lambda2 = [this](){ return i + 42; };

    i = 42;

    int a = lambda1(); // a = 42
    int b = lambda2(); // b = 84
  }
};
```

[Slide 341] `std::function`[4]

- `std::function`: general-purpose wrapper for all callable targets
- Allows storing, copying, and calling the wrapped target
- Often adds considerable overhead $\rightsquigarrow$ **avoid where possible**

```
#include <functional>

std::function<int()> getFunction(bool first) {
  int a = 14;
  if (first)
    return [=]() { return a; };
  else
    return [=]() { return 2 * a; };
}

int main() {
  return getFunction(false)() + getFunction(true)(); // 42
}
```

[Slide 342] Passing Function Objects as Parameters

```
#include <functional>

int bad(int (*fn)()) { return fn(); }
int slow(const std::function<int()>& fn) { return fn(); }
template <typename Fn> int good(Fn&& fn) { return fn(); }

struct Functor {
    int operator()() { return 42; }
};
int main() {
    Functor ftor;
    bad([]() { return 42; }); // OK
    bad(ftor); // ERROR!
```

[4] https://en.cppreference.com/w/cpp/utility/functional/function

```
    slow([]() { return 42; }); // OK
    slow(ftor); // OK
    good([]() { return 42; }); // OK
    good(ftor); // OK
}
```

> Usually, code that intends to call function objects should rely on templates. Function pointers severely limit the usability of the function. `std::function` is more flexible, but is quite expensive to create and use.

Quiz: Have you filled out the lecture evaluation?

A. Yes!
B. Of course!
C. Not yet, but I really, *really* promise to do it later today.

9.2. Algorithms

[Slide 344] Algorithms Library[5]

- C++ standard library provides several functions for sorting, searching, etc.
- Operations on ranges of elements `[begin, end)`
  - Operate on appropriate iterator types (incl. pointers)
- Mostly in `<algorithm>`, but also `<numeric>`, `<memory>`, `<cstdlib>`

[Slide 345] `std::sort`[6]

- Sort elements in range `[begin, end)` on *RandomAccessIterators*
- Elements must be swappable, move-assignable and move-constructible
- $\mathcal{O}(n \log n)$ comparisons, not stable

```
#include <algorithm>
#include <vector>

int main() {
  std::vector<unsigned> v = {3, 4, 1, 2};
  std::sort(v.begin(), v.end()); // 1, 2, 3, 4
}
```

[Slide 346] Custom Comparison[7]

- Comparator supplied as functor: `bool cmp(const T&, const T&)`
- Must establish strict weak ordering: `true` iff `a < b`
  - `!(a < a)`, `a < b` ⇒ `!(b < a)`, `a < b && b < c` ⇒ `a < c`, `!(a < b) && !(b < a)` ⇒ `a` ≈ `b`

[5] https://en.cppreference.com/w/cpp/algorithm
[6] https://en.cppreference.com/w/cpp/algorithm/sort
[7] https://en.cppreference.com/w/cpp/named_req/Compare

```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
}
```

[Slide 347] Other Sorting Operations

- `std::sort` is unstable (order of equal-ranked elements not maintained)
- `std::stable_sort` is stable
- `std::partial_sort` to find smallest n elements
  - More efficient if only top-k elements are interesting
- `std::is_sorted` to check whether a range is sorted
- `std::is_sorted_until` to find first unsorted element
- `std::partition` to reorder elements based on result of predicate
- Some others, see reference

[Slide 348] Searching – Unsorted[8]

- Find first element, returns iterator
  - `std::find`, `std::find_if`, `std::find_if_not`
- Count matching elements: `std::count`/`std::count_if`
- Search for a range of elements: `std::search`
- Check if condition holds: `any_of`, `all_of`, `none_of`
- Many more operations, see reference

[Slide 349] Searching – Unsorted

```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());

  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());
}
```

[8] https://en.cppreference.com/w/cpp/algorithm#Non-modifying_sequence_operations

[Slide 350] Searching – Sorted

- On sorted ranges, binary search is more efficient
- $\mathcal{O}(\log n)$ for *RandomAccessIterator*
  - $\mathcal{O}(n)$ when called with *ForwardIterator*!
- `std::binary_search` – check whether element is contained
- `std::lower_bound` – iterator to first element $\geq$ search value
- `std::upper_bound` – iterator to first element $>$ search value
- `std::equal_range` – pair of `lower_bound` and `upper_bound`

[Slide 351] Searching – Sorted

```
#include <algorithm>
#include <vector>
#include <cassert>
int main() {
  std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};
  assert(true == std::binary_search(v.begin(), v.end(), 3));
  assert(false == std::binary_search(v.begin(), v.end(), 0));

  assert(v.begin()+3 == std::lower_bound(v.begin(), v.end(), 3));
  assert(v.begin() == std::lower_bound(v.begin(), v.end(), 0));
  assert(v.begin()+6 == std::upper_bound(v.begin(), v.end(), 3));
  assert(v.end() == std::upper_bound(v.begin(), v.end(), 4));
}
```

[Slide 352] Searching and Sorting

- Sort + binary search useful if:
  - Separated insert and lookup phases
  - Many searches are required
- Sorting might not be a good idea if:
  - Order cannot be changed and would need to make copy
  - There are frequent updated or insertions

[Slide 353] Permutations[9]

- Functions for iterating over permutations in lexicographical order
- `std::next_permutation`
  - `false` if permutation was the last permutation (sorted in descending order)
- `std::prev_permutation`
  - `false` if permutation was the last permutation (sorted in ascending order)

```
#include <algorithm>
#include <vector>
#include <cassert>
int main() {
```

[9] https://en.cppreference.com/w/cpp/algorithm/next_permutation

```
  std::vector<int> v = {1, 2, 3};
  std::next_permutation(v.begin(), v.end()); // true, v == {1, 3, 2}
  std::next_permutation(v.begin(), v.end()); // true, v == {2, 1, 3}
  std::prev_permutation(v.begin(), v.end()); // true, v == {1, 3, 2}
  std::prev_permutation(v.begin(), v.end()); // true, v == {1, 2, 3}
  std::prev_permutation(v.begin(), v.end()); // false, v == {3, 2, 1}
}
```

[Slide 354] Additional Functionality

- `std::min_element`/`std::max_element` – operate over range of elements
- `std::merge`/`std::inplace_merge`
- `std::copy` – copy elements
- Many set operations, sampling, heap operations, ...
- `std::iota` – initialize with increasing values

```
#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}
}
```

9.3. Ranges

[Slide 355] Ranges[10]

- Ranges provide an abstraction of iterator pairs seen so far
- Views of ranges can be manipulated through adaptors

```
#include <ranges>
#include <print>
#include <map>
int main() {
  std::map<int, int> map{{10, 22}, {1, 2}, {3, 4}};
  for (auto key : map | std::views::keys)
    std::println("{}", key);
  // Prints: 1 3 10
}
```

[Slide 356] Range Factories

- Most containers can be used directly as ranges
  - Details specified in range concepts, `ranges::range` and `ranges::viewable_range` (for ranges convertible into view for further transformation)
- Range factories: create commonly used views without dedicated container
  - `views::empty`, `views::single`, `views::iota`

[10] `https://en.cppreference.com/w/cpp/ranges`

```
#include <ranges>
#include <print>
int main() {
  for (auto i : std::views::iota(1, 5))
    std::println("{}", i);
  // Prints: 1 2 3 4
}
```

[Slide 357] Range Adaptors

- (Lazily) Transform elements of range, return a view
- Might take additional arguments for transformation
- Can be chained, either by $C_2(C_1(R))$ or `R | C1 | C2`

```
#include <ranges>
#include <print>
#include <map>

int main() {
  std::map<int, int> map{{1, 2}, {3, 4}};
  for (auto key : (map | std::views::keys | std::views::reverse))
    std::println("{}", key);

  auto square = [](auto x) { return x * x; };
  for (auto sq : (map | std::views::keys | std::views::transform(square)))
    std::println("{}", sq);
}
```

[Slide 358] Range Adaptors

Quiz: What is the output of the program?

```
#include <print>
#include <ranges>
#include <vector>
int fn() {
  auto print = [](int x) { std::print("{}␣", x); return x; };
  std::vector<int> vec{1, 2, 3, 4, 5, 6};
  auto v = vec | std::views::reverse | std::views::transform(print)
         | std::views::drop(2);
  return *v.begin();
}
int main() {
  std::println("{}", fn());
}
```

A. (compile error) B. `6 5 4 3 2 1 4` C. `6 5 4 4` D. `4 4`

9.4. Random Number Generators

[Slide 359] Random Number Generators[11]

- C++ standard library defines pseudo-random number generators/distributions
- PRNGs can (and should) be seeded; not thread-safe
- Example: `std::mt19937` (32-bit)/`std::mt19937_64`

```
#include <cstdint>
#include <random>
int main() {
  std::mt19937 engine(42); // seed = 42
  unsigned a = engine(); // a == 1608637542
  unsigned b = engine(); // b == 3421126067
}
```

> Some implementations of `rand` produce very low quality random numbers. The Mersenne Twister (as implemented by `std::mt19937`) often provides much better random numbers and is reasonably efficient.

[Slide 360] Other Random Number Generators

- `std::mt19937` provides typically good enough pseudo-randomness
- `std::random_device` provides *true* randomness
  - Typically rather slow, might degrade to pseudo-randomness if no entropy is available
  - Not good for testing, where determinism is wanted
  - Typical use: get seed `std::mt19937 engine(std::random_device()());`
- `std::default_random_engine` – implementation-defined
  - Non-portable
- `rand()` from `<cstdlib>`
  - Quality of random numbers often rather bad
  - Avoid, strongly prefer C++ random functionality

[Slide 361] Distributions

- Random number generators have fixed output range, approximately uniform
- Distributions transform output of RNG
  - Uniform, normal, Bernoulli, Possion, ...

```
#include <random>
int main() {
  std::mt19937 engine(42);
  std::uniform_int_distribution<int> dist(-2, 2); // range [-2, 2]
  int d1 = dist(engine); // d1 == -1
  int d2 = dist(engine); // d2 == -2
}
```

[11] https://en.cppreference.com/w/cpp/header/random

[Slide 362] Remainder/Modulo for Uniform Distributions

Quiz: What is problematic about this function?

```
unsigned genRand8() { /* perfect RNG for 3 bits of randomness */ }
unsigned rollDice() {
  return genRand8() % 6 + 1;
}
```

A. Integer remainder is a very slow operation.
B. Some values are more likely than others.
C. There is no problem.

Here is an unproblematic alternative implementation:

```
#include <random>
int main() {
  // Use random device to seed generator
  std::random_device rd;
  // Use pseudo-random generator to get random numbers
  std::mt19937 engine(rd());
  // Use distribution to generate dice rolls
  std::uniform_int_distribution<> dist(1, 6);
  int d1 = dist(engine); // gets random dice roll
  int d2 = dist(engine); // gets random dice roll
}
```

[Slide 363] Algorithms, Functions, and Lambdas – Summary

- Function pointers can refer to functions, but have no state
- Member function pointers can refer non-static member functions
- Functors are a concept representing callable objects
- Lambdas are unnamed structures that can capture values
- Several algorithms are provided in the standard library
- Ranges provide an abstraction for iterator pairs
- Ranges can be transformed, creating views
- (Pseudo) random number generators and distributions are provided

[Slide 364] Algorithms, Functions, and Lambdas – Questions

- What are requirements for a type to be a function object?
- What is the type of a lambda expression?
- Where are lambda captures stored?
- When can by-reference captures be problematic?
- How to write functions that take functors as argument?
- How to find the insertion point for some value into a sorted array?
- Why is modulo for random numbers generally not a good idea?

10. Exceptions and Advanced Memory Management

10.1. Exceptions

[Slide 366] C++ Exceptions[1]

- Exceptions have similar semantics as in other languages

⇒ Transfer control and propagate information up the call stack

- Thrown by `throw`, `new`, and some standard library functions
- Exceptions can be handled in `try`–`catch` blocks
- Unhandled exceptions lead to termination
- When transferring control up the call stack, the runtime performs *stack unwinding*
- All objects with automatic storage duration are destructed

⇝ Correct behavior of RAII classes

[Slide 367] Throwing Exceptions[2]

- `throw` *expression* `;`
- Objects of any complete type can be thrown
- Exception object (heap-allocaetd) copy-initialized with expression
- Typically a subclass of `std::exception`

```
#include <exception>
void foo(unsigned i) {
  if (i == 42)
    throw 42;

  throw std::exception();
}
```

[Slide 368] Handling Exceptions[3]

- `try { ... } catch (`*declaration*`) { ... };`
- Exceptions occuring during `try`-block can be handled in `catch`-block
- Declaration type determines which type of exception is caught

```
#include <exception>
void bar() {
  try {
```

[1] https://en.cppreference.com/w/cpp/language/exceptions
[2] https://en.cppreference.com/w/cpp/language/throw
[3] https://en.cppreference.com/w/cpp/language/catch

```
    foo(42);
  } catch (int i) { // handle exception of type int
  } catch (const std::exception& e) { // handle exception of type std::exception
  } catch (...) { // catch-all
  }
}
```

[Slide 369] Exceptions: Example

Quiz: What is problematic about this code?

```
#include <memory>
#include <print>
int foo(const int& x) { return x != 0 ? throw x : x; }
int bar(int x) { std::unique_ptr<int> ui(new int);
     *ui = x * 2; return foo(*ui); }
int main() {
  try { std::print("ok!␣{}\n", bar(21));
  } catch (int x) {}
}
```

A. Compile error: `throw` is a statement, not an expression.
B. Memory leak: Memory from `new` is leaked on exception.
C. Unhandled exception: the exception has type `const int&`.
D. Nothing: the program terminates with exit code zero.

Solution on page 189.

[Slide 370] Exceptions: Miscellaneous

- In a catch block, the current exception can be re-thrown
  - Syntax: `throw;`
  - E.g., to clean up resources and propagate exception further
- Functions can be marked as `noexcept`
  - Part of the function type
  - Indicates that the function will never throw an exception
  - Any exceptions that would propagate cause program termination
- Destructors, move constructors/assignment must not throw exceptions

Technically, the last point is not quite accurate. Destructors are `noexcept` by default, but can theoretically also throw exceptions (by marking them as `noexcept(false)`). This is *highly* problematic: if the destructor gets called during exception handling, the program terminates, because no two exceptions can be handled at the same time.

Move constructors/assignments should be explicitly marked as *noexcept*. Otherwise, some containers that guarantee strong exception guarantees must resort to less efficient copying (e.g., growing a vector either succeeds or does nothing when one element could not be copied, but will never leave the vector in a corrupt state).

[Slide 371] Exceptions: Constructors

Quiz: Which answer is correct?

```
#include <print>
struct A { A() { throw 1; } };
struct B {
    A a;
    B() try : a() {
    } catch (int x) {
        std::println("whoops? {}", x);
        throw; // rethrow exception
    }
};
int main() { try { B b; } catch (int x) { return x; } }
```

A. Compile error: Cannot use `try` outside function body.
B. The `throw;` is not necessary.
C. `a` is life in the `catch` block of the constructor.
D. No object of type `A` can be constructed, but objects of type `B` can be.

Solution on page 189.

[Slide 372] Exceptions: Performance and Code Size Considerations

- Exception handling (stack unwinding) is rather expensive
- Low overhead if no exceptions are thrown

⇒ In any case, exceptions should be used rarely

- The mere *possibility* of exceptions inhibits some optimizations
  - Increased control flow complexity, more state must be kept in stack memory
- For every possibly throwing call, corresponding cleanup code must be generated
- Unwind tables that map code location to cleanup landing pad can grow large

⇝ *Enabling* exceptions can have substantial code size impact

  - To disable exceptions: `-fno-exceptions`

[Slide 373] Exceptions: Guidelines

- Use exceptions only in rare cases
- E.g., dynamic runtime errors (e.g., malformed data)
- Do not use exceptions for programmer errors
  - Use assertions for this
- Do not use exceptions for control flow
  - Use regular control flow operations for this
- Generally: exceptions should be **avoided** where possible
- When not using exceptions at all, disable them via a compiler flag

> The increased code size together with the increased code complexity and reduced readability are the reason for some projects/companies to outright ban C++ exceptions.

10.2. Explicit Object Construction

[Slide 374] `operator new`

- `operator new` (`<new>`) can take arguments[4]
- Default, implicitly: `operator new (size)`
- Example: overload with extra arg `std::nothrow_t`

```
#include <new>
#include <array>
#include <print>
struct A { /* ... */ };
int main() {
  // Will throw std::bad_alloc
  auto* p1 = new std::array<int, 100000000000>();
  // Will return nullptr on allocation failure
  auto* p2 = new(std::nothrow) std::array<int, 100000000000>();
  if (!p2)
    std::println("allocation␣failed!");
}
```

[Slide 375] Manually managing memory

- Sometimes, the default memory management operations are not enough
  - E.g., repeatedly calling `new` (explicit or implicit) is too expensive
  - E.g., for reusing already available memory

⇝ Placement new: construct object in already allocated storage

- Manually call constructor and destructor

[Slide 376] Placement `new`

- `operator new(size, void* ptr)`
  - Returns `ptr` without doing any allocation
- Alignment must be ensured manually

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

[4] https://en.cppreference.com/w/cpp/memory/new/operator_new

[Slide 377] Placement `new` and Lifetime

- Placement new ends lifetime of overlapping objects; creates new object
- Lifetime is nested within the underlying storage

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

[Slide 378] How to Deallocate?

Quiz: How to deallocate `s1`? What to write instead of `XXX`?

```
template <class T, size_t N>
class TAlloc {
  alignas(T) std::byte buffer[sizeof(T[N])];
  size_t cnt = 0;
public:
  T* make(T&& t) {
    void* vp = &buffer[sizeof(T)*cnt++];
    T* r = reinterpret_cast<T*>(vp);
    ::new(r) T(std::move(t));
    return r;
  }
};
int main() {
  TAlloc<std::string, 3> ta;
  auto* s1 = ta.make("Hello␣World!");
  // XXX
}
```

A. `delete(s1);`
B. `s1->~string();`
C. `s1->~basic_string();`
D. `ta.~TAlloc();`
E. Nothing, the strings are automatically freed at the end of `main`.

Solution on page 189.

> A typical implementation would destruct all allocated instances in the destructor of `TAlloc`.

[Slide 379] Placement new with `unique_ptr`

- `std::unique_ptr<T, Deleter>` – specify type of deleter
- Second parameter in constructor to specify deleter instance
- Default deleter calls `delete`

- For use with non-standard allocation, a custom deleter is required
- Code that uses custom allocators is typically rather complex ⇒ `unique_ptr` is often not particularly useful in such contexts

[Slide 380] Overloading `operator new`

- Classes can overload `operator new` and `operator delete`
- Can also provide overloads with extra arguments
- Rarely useful, e.g.:
  - Allocating extra storage after/before the object

10.3. Unions

[Slide 381] `union`

- Class type that holds only one of its non-static members at a time
- Storage large enough to hold largest element
- All data members have the same address
- Writing to a union member *activates* it
- Reading an inactive union member is undefined behavior

```
union MyUnion { float f; long l; short a[2]; };
static_assert(sizeof(MyUnion) == sizeof(long));
int main() {
  MyUnion u; // f active, default-initialized
  u.f = 123.0; // f active
  u.a[1] = 12; // a active
  return u.a[1]; // ok
}
```

[Slide 382] Union: Example

Quiz: What is the output of the program?

```
#include <print>
int main() {
  using Converter = union { float f; unsigned u; };
  std::println("{:08x}", Converter{32.5f}.u);
  return 0;
}
```

A. Compile error: Cannot have untyped union.
B. Compile error: Union initializer is ambiguous.
C. Undefined behavior: Program reads inactive union member.
D. The integer representation of `32.5f` (`42020000`).

Solution on page 189.

[Slide 383] `std::bit_cast`[5]

- For bitwise reinterpretation of object representations, use `std::bit_cast<TargetTy>()` from `<bit>`
  - Do not use `union` for this – C++ differs from C here
  - Do not use `reinterpret_cast`

[Slide 384] Union with Non-Primitive Types

- `union`s can have non-primitive members
- `union` doesn't know which member is active...
- Lifetime needs to be managed explicitly outside of the union
- Typical use as part of a struct which tracks active element
- Can be used to implement more efficient variant
- Very difficult to get right

⇝ Prefer `std::variant`

[Slide 385] Union with Non-Primitive Types: Example

```
union U {
  std::vector<int> v;
  std::string s;
  // needs explicit destructor -- can't do anything!
  // union doesn't know which member is active
  ~U() {}
};
int main() {
  U u{}; // constructs first element
  u.v.push_back(123);
  u.v.~vector<int>(); // lifetime of u.v ends
  new(&u.s) std::string("123"); // lifetime of u.s begins
  std::println("{}", u.s);
  u.s.~basic_string(); // lifetime of u.s ends
  // ~U() will be called, but is defined to do nothing
}
```

10.4. Implementing a Vector

[Slide 386] Implementing our own Vector

- At this point, we can implement our own vector

(see script)

```
template <class T>
class Vec {
  T* ptr = nullptr;
```

[5] https://en.cppreference.com/w/cpp/numeric/bit_cast

```
  size_t sz = 0;
  size_t cap = 0;

public:
  using value_type = T;
  using size_type = size_t;
  using difference_type = ptrdiff_t;
  using reference = T&;
  using const_reference = const T&;
  using pointer = T*;
  using const_pointer = const T*;
  using iterator = T*;
  using const_iterator = const T*;

  Vec() = default;

  ~Vec() {
    reset();
  }

  Vec(const Vec&) = delete;
  Vec(Vec&& other) noexcept {
    *this = std::move(other);
  }

  Vec& operator=(const Vec&) = delete;
  Vec& operator=(Vec&& other) noexcept {
    if (&other == this)
      return *this;
    reset();
    ptr = other.ptr;
    sz = other.sz;
    cap = other.cap;
    other.ptr = nullptr;
    other.sz = 0;
    other.cap = 0;
    return *this;
  }

  pointer data() { return ptr; }
  const_pointer data() const { return ptr; }
  size_type size() const { return sz; }
  size_type capacity() const { return cap; }
  bool empty() const { return !size(); }

  reference operator[](size_type idx) { return data()[idx]; }
  const_reference operator[](size_type idx) const { return data()[idx]; }
  reference front() { return data()[0]; }
  const_reference front() const { return data()[0]; }
  reference back() { return data()[size() - 1]; }
  const_reference back() const { return data()[size() - 1]; }

  iterator begin() { return data(); }
  iterator end() { return data() + size(); }
```

```
  const_iterator begin() const { return data(); }
  const_iterator end() const { return data() + size(); }
  const_iterator cbegin() const { return data(); }
  const_iterator cend() const { return data() + size(); }

  void push_back(const T& v) {
    if (size() == capacity())
      grow((capacity() + 1) * 2);
    new(end()) T(v);
    sz++;
  }
  void push_back(T&& v) {
    if (size() == capacity())
      grow((capacity() + 1) * 2);
    new(end()) T(std::move(v));
    sz++;
  }
  void pop_back() {
    back().~T();
    sz--;
  }

  void clear() {
    for (pointer p = begin(); p != end(); ++p)
      p->~T();
    sz = 0;
  }

  void reset() {
    if (ptr) {
      clear();
      free(ptr);
      ptr = nullptr;
      cap = 0;
    }
  }

private:
  void grow(size_type newCap) {
    // NB: this could be wrong due, T might require more alignment.
    T* newPtr = static_cast<T*>(malloc(newCap * sizeof(T)));
    for (pointer p = begin(), pn = newPtr; p != end(); ++p, ++pn) {
      new(pn) T(std::move(*p));
      p->~T();
    }
    free(ptr);
    ptr = newPtr;
    cap = newCap;
  }
};

struct Demo {
  // Always points to this, *not* updated on copy/move!
  // We allocate this pointer on the heap and use LeakSanitizer
```

```
  // to identify objects that were not freed.
  Demo** check;
  int val;

  Demo(int val) : check(new Demo*(this)), val(val) {}
  ~Demo() {
    assert(this == *check);
    delete check;
  }

  Demo(const Demo&) = delete;
  Demo(Demo&& other) : check(new Demo*(this)) {
    val = other.val;
  }
  Demo& operator=(const Demo&) = delete;
  Demo& operator=(Demo&& other) {
    val = other.val;
    return *this;
  }
};

int main() {
  Vec<Demo> v, w;
  assert(v.empty());
  v.push_back(12);
  assert(v.size() == 1);
  assert(v[0].val == 12);
  w.push_back(14);
  assert(w.size() == 1);
  assert(w[0].val == 14);

  size_t oldCap = v.capacity();
  w = std::move(v);
  assert(v.empty());
  assert(v.capacity() == 0);
  assert(w.size() == 1);
  assert(w.capacity() == oldCap);
  assert(w[0].val == 12);
  w.push_back(42);
  w.push_back(20);
  w.push_back(21);
  w.push_back(22);
  w.push_back(23);
  w.pop_back();
  assert(w.size() == 5);

  Vec<Demo> x(std::move(w));
  assert(w.empty());
  assert(w.capacity() == 0);
  assert(x.size() == 5);
  assert(x[0].val == 12);
  assert(x[1].val == 42);
}
```

[Slide 387] Allocating Raw/Uninitialized Memory

- C `malloc`/`free` often work, but not always
- Problem: type might have increased alignment requirement
- `std::allocator<T>`[6] respects additional requirements
  - `allocate(elementCount)` – allocate an array suitable for n objects
  - `deallocate(ptr, elementCount)` – deallocate previously allocated memory

Providing the size for the deallocation allows for a more efficient implementation of allocators, as they don't have to track the size of allocated memory regions themselves.

We can adjust our vector to use an allocator:

```
template <class T>
class Vec {
  // ...
  [[no_unique_address]] std::allocator<T> alloc;

  // ...
  // ... replace malloc/free with allocate/deallocate, e.g. as in:
  void grow(size_type newCap) {
    T* newPtr = alloc.allocate(newCap);
    for (pointer p = begin(), pn = newPtr; p != end(); ++p, ++pn) {
      new(pn) T(std::move(*p));
      p->~T();
    }
    alloc.deallocate(ptr, cap);
    ptr = newPtr;
    cap = newCap;
  }
};
```

[Slide 388] Helper Functions for Handling Uninitialized Memory

- Provides more guarantees in case of an exception
- `std::uninitialized_move` – move range of elements into uninitialized memory
- `std::uninitialized_default_construct` – default-construct range of elements into uninitialized memory
- `std::destroy` – destruct range of elements

[Slide 389] Exception Safety when Moving

- Move constructor/assignment might throw exceptions

Quiz: (Why) is this problematic?

A. Afterwards, vector might be in unrepairable state
B. Exception cannot be caught properly
C. New allocation will always be leaked

[6] `https://en.cppreference.com/w/cpp/memory/allocator`

D. This is not a problem, just annoying

Solution on page 190.

- `std::vector` guarantees exception safety
  - E.g., `push_back` guarantees to have no effect if any operations throws
- If move operations are not `noexcept`, elements will be copied instead

[Slide 390] `memcpy/memmove`

- For primitive data types, constructing/destructing is not required
- `std::is_trivially_copyable_v<T>` – indicates whether byte-wise copying is possible
  - In fact, this is also possible for structs of trivially copyable types
- `std::memcpy(dest, src, count)` – copy bytes between non-overlapping regions
- `std::memmove(dest, src, count)` – copy bytes between regions
- In both cases, alignment of destination must be suitable

10.5. Custom Allocator Functions

[Slide 391] Custom Allocators

- Sometimes, the default allocator is not good enough
  - Many small allocations are expensive
  - All allocations have to be freed separately
  - Every allocation has memory overhead (e.g., tracking allocation size)
  - Requires synchronization in multi-threaded applications
  - Possibly bad locality
- Typical solution: bump pointer allocator
  - Allocate large chunk of memory once
  - Hand out slices for individual allocations
  - Free allocated memory when allocator is destroyed

[Slide 392] Custom Allocators in C++

- Requirements specified by *Allocator*
  - In essence: `value_type`, `allocate`, `deallocate`
- Containers are allocator-aware and can use custom allocators
- Bump-ptr allocator in C++ standard library: `std::pmr::monotonic_buffer_resource`
  - Usable with `std::pmr::polymorphic_allocator` as allocator
  - Performance characteristics not that good (see inheritance later)
- For performance with many small allocations, custom allocators are often required

[Slide 393] Exceptions and Advanced Memory Management – Summary

- C++ Exceptions allow for unordinary control flow transfers
- Almost everything can be thrown and caught
- Exception unwinding calls destructors of objects with automatic storage duration
- Objects can be constructed in allocated memory with placement new
- Required when memory allocation and object construction are separated
- `union`s provide an untagged overlapping storage
- Writing exception-safe code is difficult
- Custom allocators can substantially improve performance in some applications

[Slide 394] Exceptions and Advanced Memory Management – Questions

- Why do some people see C++ exceptions as problematic?
- What are upsides and downsides of C++ exceptions?
- Why is writing exception-safe code difficult?
- What happens when an exception is thrown in a `noexcept` function?
- Why should move constructors/assignment be marked as `noexcept`?
- What requirements must be met for placement new?
- Why is using `union` much more difficult than in C?
- What are benefits of bump pointer allocators?

11. Compile-Time Programming

11.1. Attributes

[Slide 396] Attributes[1]

- Almost everything can be annotated with attributes
- C++-style attributes: `[[ <attribute> ]]`
  - Parenthesis inside attributes must be balanced, unknown attributes ignored
- Preprocessor `__has_cpp_attribute(name)` to query support

```
#include <cassert>
int foo();
int foo(int z) {
    // Variable attribute: suppresses warning about unused variable
    int x = 5, y [[maybe_unused]] = foo();
    assert(y); // <-- in release builds, y is unused
    if (z > 10) [[likely]] // Give hint that condition is likely
        x += z * z;
    return x;
}
```

[Slide 397] Function Attributes

- `[[nodiscard]]` – cause warning when function result is unused
  - Beneficial to enforce error handling etc.
- `[[deprecated(reason)]]` – cause warning when function is used
- `[[noreturn]]` – indicate that function does not return

```
#include <cassert>
[[nodiscard, deprecated("use␣xyz␣instead")]] int oldFunc();
// Second attribute is unknown and ignored, causes warning
[[noreturn, unknown_and_ignored]] void myExit();
int foo(int z) {
    oldFunc();
    myExit();
    // no warning about missing return in non-void function
}
```

[Slide 398] Implementation-Defined Attributes

- Most attributes are implementation-defined
  - E.g., Clang[2] and GCC support hundreds of attributes

[1] https://en.cppreference.com/w/cpp/language/attributes
[2] https://clang.llvm.org/docs/AttributeReference.html

Some examples:

- `[[gnu::always_inline]]` – Always inline function when possible
- `[[clang::optnone]]` – Disable optimization for specific function
  - E.g., to ease debugging of a single function
- `[[clang::musttail]]` – Return statement must make tail call
  - Tail call = no stack frame growth for function call, useful for tail-recursive functions

> Typically, attributes are wrapped as macros if supported by the compiler. This enables to use non-standard attributes, which have different spellings in different compilers. Example:
> ```
> #if __has_cpp_attribute(clang::foo)
> #define FOO [[clang::foo]]
> #elif __has_cpp_attribute(msvc::foo)
> #define FOO [[msvc::foo]]
> #else
> #define FOO
> #endif
> ```

[Slide 399] `[[clang::lifetimebound]]`

- Indicate that return value may refer to parameter object
- Causes warning when parameter's lifetime is shorter than returned value

```
#include <print>
#include <format>
#include <string>
#include <string_view>
struct Error {
    std::string_view msg;
    Error(const std::string &msg [[clang::lifetimebound]]) : msg(msg) {}
};
int main() {
    // Construct error with temporary string...
    Error err(std::format("foo!␣{}", 123));
    // Warning: dangling reference
    std::println("error:␣{}", err.msg);
}
```

[Slide 400] `[[clang::lifetimebound]]` – Example

Quiz: Which parameter/function should get the attribute?

```
struct Foo {
    std::string msg;
    explicit Foo(const std::string& msg) : msg(msg) {}
    void addMsg(const std::string& add) { msg += add;}
    const std::string& getMsg() const { return msg; }
};
```

A. `msg`
B. `msg` and `add`
C. `getMsg` (before block, due to `this`)
D. `getMsg`, `msg`, and `add`

Solution on page 190.

[Slide 401] GNU-Style and MSVC-Style Attributes

- GNU-style: `__attribute__((attrs))`
- MSVC-style: `__declspec(attrs)`
- Much older (⇝ more widely used) than C++ attributes
- Syntax of attributes sometimes slightly different (see manual)
- Prefer C++-style attributes when possible

> However, do note that while many GNU attributes can be used in C++-style attributes with the `gnu::` prefix, a few attributes are only supported with the GNU attribute syntax.

11.2. Compile-Time Programming

[Slide 402] Constant Expressions

- Certain language constructs require compile-time constants
  - E.g., array bounds, non-type template parameters, bit-field length, `static_assert`, enum values, ...
- So far limited to constant literals or simple expressions

```
static int return4() { return 4; }
int main() {
    const int x1 = 4;
    std::array<int, x1 + 3> arr1; // ok... (due to exception in standard)
    const int x2 = return4();
    std::array<int, x2> arr2; // Error! x2 is not a constant expression
}
```

- `const` just marks a variable as non-modifiable

[Slide 403] `constexpr`[3]

- `constexpr` variables – can be used as constant expressions
  - Must be initialized immediately with a constant expression
  - Implicitly `const`; some type restrictions (see reference)
- `constexpr` functions – function that is evaluatable at compile-time
  - Result can be used as constant expression
  - Destructor must be `constexpr` (or trivial)

[3] `https://en.cppreference.com/w/cpp/language/constexpr`

[Slide 404] `constexpr` – Example

```
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 = g(x); // ok
    constexpr int yy = y; // ERROR: f(x) not constant expression
    constexpr int zz = z; // ok
}
```

[Slide 405] `constexpr` – Example

Quiz: Which statement is correct?

```
constexpr int* f(int n) { return n ? new int(n) : nullptr; }
int* f1(int n) { return f(n); }
int* f2(int n) { constexpr auto r = f(0); return r; }
int* f3(int n) { constexpr auto r = f(n); return r; }
constexpr int* f4(int n) { return f(n); }
```

A. `f`: heap allocation is performed at compile-time
B. `f2`: `f(0)` is not a constant expression
C. `f3`: `f(n)` is not a constant expression
D. `f4`: must return constant expression

Solution on page 190.

[Slide 406] `constexpr` vs. `consteval` Functions

- `constexpr` functions *can* be evaluated at compile-time
  - Implicitly `inline`
  - When constant expression is required, must yield compile-time constant
  - Other code paths can work with non-compile-time constants
  - Can be called at runtime with dynamic values
- `consteval` functions *must* be evaluated at compile-time
  - Implicitly `inline`
  - *Every* function call must yield compile-time constant
  - Cannot be mixed with `constexpr`

[Slide 407] `consteval` – Example

```
int sqr(int n) { return n * n; }
consteval int sqrConsteval(int n) { return n * n; }
constexpr int sqrConstexpr(int n) { return n * n; }
```

```
int main() {
    constexpr int p1 = sqr(100); // ERROR: not constexpr
    constexpr int p2 = sqrConsteval(100);
    constexpr int p3 = sqrConstexpr(100);
    int x = 100;
    int p4 = sqr(x);
    int p5 = sqrConsteval(x); // ERROR: x not constant expression
    int p6 = sqrConstexpr(x);
    int p7 = sqrConsteval(100); // compile-time
    int p8 = sqrConstexpr(100); // run-time or compile-time
}
```

[Slide 408] `constexpr`/`consteval` and Compile-Time Execution

Quiz: Which statement is correct?

A. Non-`constexpr`/`consteval` functions are always evaluated at runtime.
B. When possible, `constexpr` functions are evaluated at compile-time.
C. `consteval` functions can only include compile-time evaluable code.
D. `constexpr` functions can be defined in headers without ODR violations.

Solution on page 190.

[Slide 409] Compile-Time Evaluation – Restrictions

- No undefined behavior (compile-time error)
- Calling functions that are only declared
- Accessing `volatile` variables
- Dynamic memory allocations that are not *delete*-ed in same expression
- Placement `new` (lifted in C++26)
- No `reinterpret_cast`
- Implementation-defined restrictions
- See reference for the full list[4]

[Slide 410] `consteval` – Example

Quiz: What is problematic about this code?

```
#include <vector>
consteval int fib(int n) {
    std::vector<int> i{0, 1};
    while (i.size() <= n)
        i.push_back(i[i.size() - 2] + i.back());
    return i[n];
}
static_assert(fib(6) == 8);
```

A. Cannot use `std::vector` in `consteval` function

[4] `https://en.cppreference.com/w/cpp/language/constant_expression`

B. Cannot use dynamic memory allocation in `consteval` function
C. `fib(6)` $= 13 \neq 8$
D. Nothing, the code compiles without errors

Solution on page 190.

A typical use-case of `consteval` is to pre-compute lookup tables, static (perfect) hash tables, etc. during compilation for efficient execution at runtime. As an example:

```
template <unsigned N>
consteval std::array<unsigned, N> computeFib() {
    std::array<unsigned, N> res{0, 1};
    for (unsigned i = 2; i < N; ++i)
        res[i] = res[i - 1] + res[i - 2];
    return res;
}

int fib(unsigned n) {
    static constexpr auto cache = computeFib<1000>();
    return cache[n];
}
```

[Slide 411] `if constexpr`[5]

- `if constexpr (...)` – compile-time if
- Not-taken code paths are *discarded*
  - For templates where condition depends on template parameters: not instantiated
- Benefit over `#if`: syntax and large parts of semantics are checked

```
// Example from cppreference
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
}
```

Due to the semantic checks of disabled code paths, `if constexpr` should be preferred over macros. The disabled path of macro-based condition never reaches the compiler and, as developers tend to not thoroughly test all configurations, is more likely to be non-functional.

Usage examples can include conditionally enable debug checks and type-dependent code paths.code paths.

[Slide 412] `if consteval`

- `if consteval { ... }` – execute block if in constant-evaluated context
- `if ! consteval { ... }` – if in not constant-evaluated context

[5] `https://en.cppreference.com/w/cpp/language/if#Constexpr_if`

- `std::is_constant_evaluated()` – "legacy" C++20 mechanism

```
constexpr int compute(int x) {
    if consteval {
        // use slower, constant-evaluable algorithm
    } else {
        // use faster algorithm that cannot be executed during compilation
    }
}
```

> For example, a square-and-multiply algorithm might be used to implement an integer pow operation at compile-time, while `std::pow` can be used at runtime, as it *might* be more efficient.

[Slide 413] `constinit`[6]

- Variables with static/thread-local storage duration are either
  - ... constant-initialized, i.e., take compile-time constants
  - ... dynamically initialized, i.e., constructor called at program start-up
- `constexpr` enforces the former and prevents modifications
- `constinit` enforces the former, but allows modifications

```
constexpr int square(int n) { return n * n; }
constinit int sq5 = square(5); // mutable variable
```

> `const constinit` is not quite the same as `constexpr`, as constant destruction is not required.

11.3. decltype

[Slide 414] `decltype`[7]

- Possibly unknown types can often be deduced with `auto`
- Sometimes, knowing the exact type of an expression is useful
  - E.g., for explicitly specifying template parameters
- `decltype(`*`identifier/class member access`*`)` – yield type of entity
- `decltype(`*`expression`*`)` – yield type of expression
  - lvalue: `T&`, xvalue: `T&&`, prvalue: `T`
  - Expression is not evaluated
- Note: `decltype(x)` and `decltype((x))` are different!
  - The first yields the type of x, the latter a reference as `(x)` is an lvalue

[Slide 415] `decltype` – Examples

[6] https://en.cppreference.com/w/cpp/language/constinit
[7] https://en.cppreference.com/w/cpp/language/decltype

```
#include <concepts>

int main() {
    int x;
    const short c = 12;
    static_assert(std::same_as<decltype(x), int>);
    static_assert(!std::same_as<decltype((x)), int>);
    static_assert(std::same_as<decltype((x)), int&>);

    static_assert(std::same_as<decltype(c), const short>);
    static_assert(std::same_as<decltype((c)), const short&>);
    static_assert(std::same_as<decltype(c + c), int>); // integer promotion
}
```

[Slide 416] Interlude: Integer Promotion

- Small integer types get promoted to `int` before arithmetic is performed[8]

> Integer promotion is a relict imported from C, which originally introduced it to simplify the implementation of small data types (e.g., `char`) on platforms that could only perform operations on word-sized integers (i.e., `int`).

Quiz: Which statement is correct?

Assume `std::same_as<int, int32_t>`.

```
#include <cstdint>
constexpr mul16(uint16_t a, uint16_t b) -> auto { return a * b; }
static_assert(std::same_as<decltype(mul16(1, 1)), int>);
static_assert(mul16(0xffff, 0xffff) == 1);
```

A. The return type of `mul16` is `uint16_t`, deduced from return statement.
B. The return type of `mul16` is `unsigned`, as `uint16_t` is unsigned.
C. The second assertion fails, as it is not a constant expression.
D. The program compiles successfully.

Solution on page 190.

11.4. Template Meta-Programming

[Slide 417] Template Meta-Programming

- Templates are instantiated during compilation
- `if constexpr` makes code actually readable

```
template <unsigned N>
constexpr int templ_fib() {
    if constexpr (N <= 1)
        return N;
    else
```

[8] Simplified, but captures the important parts.

```
    return templ_fib<N-2>() + templ_fib<N-1>();
}
static_assert(templ_fib<6>() == 8);
```

[Slide 418] Template Meta-Programming, the Old Way

- Template specializations used as recursion base case

```
template <unsigned N>
constexpr int templ_fib() {
    return templ_fib<N-2>() + templ_fib<N-1>();
}

template<>
constexpr int templ_fib<0>() { return 0; }

template<>
constexpr int templ_fib<1>() { return 1; }

static_assert(templ_fib<6>() == 8);
```

> Avoid this type of meta-programming where possible.
>
> Note that template specialization makes the type system Turing-complete. Hence, plain syntactical parsing of C++ source code is an *undecidable* problem. In practice, however, this has no relevance and recursion limits are often rather small.

11.5. Concepts II

[Slide 419] Concepts

- Previously seen: type constraints with `requires` clause
  - `template <...> requires` *`bool-constant-expr`*
- Repeating requirements can be tedious

⇝ Concept = named set of requirements

```
template <class T>
concept IntOrFloat = std::integral<T> || std::floating_point<T>;
static_assert(IntOrFloat<int>);
static_assert(!IntOrFloat<int*>);

template <IntOrFloat T> T add(T a, T b) {
    return a + b;
}
```

[Slide 420] Concepts: Requirements

- For templates, the exact type is often hard to verify
- So far: "duck typing" – just assume that method/operator is available
- Concepts allow to verify that all required operations are present

```
// Parameters a,b have no storage, just used as notation for naming requirements
// NB: this is a requires expression, not a requires clause for constraints
template<typename T> concept Addable = requires (T a, T b) {
    // Verify that a + b is a valid expression.
    a + b;
};
template<typename T>
concept Graph = requires {
    // Verify that T has a member type "node_type".
    typename T::node_type;
    // ... and require that is is an integer type.
    requires std::integral<typename T::node_type>;
};
```

[Slide 421] Concepts: Requirements

Quiz: Which statement is correct?

```
template<typename T>
concept Graph = requires {
    typename T::node_type;
    requires std::integral<typename T::node_type>;
};
class MyGraph {
    using node_type = char;
};
static_assert(Graph<MyGraph>);
```

A. Syntax error: cannot use concept as boolean constant expression.
B. Assertion fails: `char` is not an integer.
C. Assertion fails: `node_type` is private.
D. The program compiles successfully.

Solution on page 190.

[Slide 422] Concepts: Compound Requirements

- We can also check the return type of an expression.

```
// Parameters a,b have no storage, just used as notation for naming requirements
template<typename T>
concept Addable = requires (T a, T b) {
    // Verify that a + b is a valid expression
    // ... and can be implicitly converted to T.
    { a + b } -> std::convertible_to<T>;

    // Alternatively:
    // ... and has the type T.
    { a + b } -> std::same_as<T>;
};
```

As a `requires` expression (not the `requires` clause!) is just a boolean expression, it is possible to write the following — although in the interest of readability, is not advisable to do so:

```
template <class T> requires requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
    { a * b } -> std::convertible_to<T>;
}
T fma(T a, T b, T c) {
    return a * b + c;
}
```

[Slide 423] Missing Requirements

- Missing requirements cause candidate to not be selected
- But: this is not an error ⇝ multiple variants can be provided

```
template <class NodeT>
concept NodeHasNumber = requires(const NodeT& n) {
    { n.getNumber() } -> std::convertible_to<unsigned>;
};
template <class NodeT>
struct NumberedGraph {
    std::unordered_map<const NodeT*, unsigned> nums;
    unsigned getNumber(const NodeT& node) requires NodeHasNumber<NodeT> {
        return node.getNumber();
    }
    unsigned getNumber(const NodeT& node) requires (!NodeHasNumber<NodeT>) {
        auto [it, inserted] = nums.try_emplace(&node, nums.size());
        return it->second;
    }
};
```

This example shows a class which provides different implementations of `getNumber` depending on the node type. If the node supports `getNumber`, that number is returned, otherwise an ad-hoc numbering in a local hash map is created.

Note that the same effect could be using `if constexpr`, which would be preferable — this example is primarily for demonstration purposes.

[Slide 424] Substitution Failure Is Not An Error (SFINAE)[9]

- If substitution of template parameters fails, the candidate is simply discarded without an error[10]
- Allows to implement `requires` in pre-C++20[11]

```
template <class T>
std::enable_if_t<std::is_integral_v<T>, T> add(T a, T b) {
    return a + b;
}
```

[9] `https://en.cppreference.com/w/cpp/language/sfinae`
[10] See reference for details
[11] `https://en.cppreference.com/w/cpp/types/enable_if`

```
template <class T>
std::enable_if_t<std::is_floating_point_v<T>, T> add(T a, T b) {
    return a + b;
}
```

- Prefer concepts if possible (i.e., code base uses C++20 or newer)

[Slide 425] Compile-Time Programming – Summary

- Attributes allow for annotation of almost all language constructs
- Most attributes are implementation-specific
- `constexpr` permits use of functions as compile-time constant expressions
- `constexpr` variables must be initialized with compile-time constant
- `consteval` functions must always be evaluated at compile-time
- `constinit` variables must have a constant initializer, but can be mutable
- Concepts can test whether certain expressions are valid
- Failing requirements or substitution failure allows for providing type-dependent implementations (or the absence thereof)

[Slide 426] Compile-Time Programming – Questions

- What happens with unsupported attributes?
- What are use cases for implementation-specific attributes?
- When are `constexpr` function calls evaluated?
- Are non-`constexpr` functions always executed at runtime?
- What is the difference between `constexpr` and `constinit`?
- What is different between `decltype(x)` and `decltype((x))`?
- Which recent C++ constructs largely eliminate the need for template metaprogramming?

12. Inheritance

[Slide 428] Object-Oriented Programming

Concepts of object-oriented programming:

- Data abstraction/encapsulation
  - ⇝ Classes in C++
- Inheritance
  - ⇝ Class derivation in C++
  - – Derived classes inherit the members of the base class
- Dynamic Binding (Polymorphism)
  - ⇝ Virtual functions in C++
  - – Derived classes can override methods of base classes
  - – By default, C++ inheritance is non-polymorphic

12.1. Non-Polymorphic Inheritance

[Slide 429] Derived Classes

- Class may be derived from one or more base classes
  - ⇝ Inheritance hierarchy
- Syntax: `class` *`class-name`* `:` *`base-specifier-list`*
- Base specifiers: `public`/`protected`/`private`; `virtual` (optional)

```
struct Base {
    int a;
};
struct DerivedA : public Base {
    int b;
};
struct DerivedB : private Base, public DerivedA {
    int c;
};
```

[Slide 430] Constructors

- Constructors of derived classes also construct base classes
  1. Direct base classes are initialized in left-to-right order
  2. Non-static data members are initialized in declaration order
  3. Constructor body is executed
- Base classes default-initialized unless specified otherwise

– Delegating constructor syntax: `Derived() : Base(arg1, arg2) {}`

[Slide 431] Constructors: Example

```
struct Base {
    Base() { std::println("Base()"); }
    Base(int) { std::println("Base(int)"); }
};
struct Derived : public Base {
    Derived() { std::println("Derived()"); }
    Derived (int a, int) : Base(a) {
        std::println("Derived(int,␣int)"); }
};
int main() {
    Derived a;
    Derived b{12, 34};
}
```

Output:

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

[Slide 432] Copy Constructors

△ **Quiz: What is the output of the program?**

```
#include <print>
struct A {
    A() { std::println("A"); }
    A(const A&) { std::println("a"); }
};
struct B : A {
    B() { std::println("B"); }
    B(const B&) { std::println("b"); }
};
int main() {
    B b1, b2(b1);
}
```

A. (compile error) B. (unspecified) C. `ABAB` D. `ABAb` E. `ABab`

Solution on page 191.

[Slide 433] Destructors

- Destructors are executed in the opposite order as constructors

◯ **Quiz: What is the output of the program?**

```
#include <print>
struct A { ~A() { std::print("A"); } };
struct B { ~B() { std::print("B"); } };
```

```
struct D : A, B { ~D() { std::print("D"); } };
int main() { D d; }
```

A. (unspecified) B. ABD C. BAD D. DAB E. DBA

Solution on page 191.

[Slide 434] Constructors: Multiple Inheritance

★ Quiz: What is the output of the program?

```
#include <print>
struct A { A() { std::print("A"); } };
struct B : A { B() { std::print("B"); } };
struct C : A { C() { std::print("C"); } };
struct D : B, C { D() { std::print("D"); } };
int main() { D d; }
```

A. (compile error) B. (unspecified) C. ABCD D. ABACD E. BACD

Solution on page 191.

[Slide 435] Unqualified Name Lookup[1]

- Names can be defined multiple times in inheritance hierarchy
- Unqualified (no `::`) lookup algorithm decides which name to choose
- Approximation: declarations in derived classes hide names from base classes

```
struct A { void a(); };
struct B : public A { void a(); void b() { a(); /* B::a() */ } };
struct C : public B {
  void c1() { a(); /* B::a() */ }
  void c2() { A::a(); /* A::a() */ } // qualified lookup
};
```

[Slide 436] Unqualified Name Lookup: Diamond Inheritance

```
struct A { void a(); };
struct B1 : public A { };
struct B2 : public A { };

struct C : public B1, public B2 {
  void c1() { a(); /* ERROR: ambiguous, a() present in B1 and B2 */ }
  void c2() { B1::a(); /* OK */ }
};
```

> In a diamond shaped inheritance hierarchy, the root class (in this example: `A`) can appear multiple times in the most derived class. This can lead to problems: not just because the names are present multiple times, also the data members are present multiple times! (This can be solved with virtual inheritance, see later.)
>
> Avoid diamond-shaped inheritance whenever possible.

[1] https://en.cppreference.com/w/cpp/language/unqualified_lookup

[Slide 437] Object Representation

- Base classes are stored as subobjects of the derived class

```
#include <cstddef>
struct A {
    int& a1;
    char a2;
};
struct B {
    short b;
};
struct C : public A, public B {
    int c;
};
static_assert(offsetof(C, a1) == 0);
static_assert(offsetof(C, a2) == 8);
static_assert(offsetof(C, b) == 10);
static_assert(offsetof(C, c) == 12);
```

> The layout rules for classes are rather complex; classes are laid out differently if they are standard-layout classes (approximately used as C structs). The exact specification can be found in the Itanium C++ ABI[a], which is widely used on many platforms.
>
> The lecture *Advanced Concepts of Programming Languages* covers the data layout of C++ classes in more detail.
>
> [a] https://itanium-cxx-abi.github.io/cxx-abi/abi.html

12.2. Inheritance Modes

[Slide 438] Inheritance Modes: `public`

- `public` inheritance: public base members become public derived members protected base members become protected derived members
- Default when derived class declared as `struct`
- Typically used to model subtyping/is-a relationship
  - Pointers/references of derived should be usable when base class is expected
  - Derived class should maintain invariants of base class
  - Derived class should not strengthen preconditions of overridden members
  - Derived class should not weaken postconditions of overridden members

[Slide 439] Inheritance Modes: `private`

- `private` inheritance: public/protected base members become private derived members
- Default when derived class declared as `class`
- Derived class can be used as base class only in derived class
- Sometimes useful
  - Mixins (e.g., special storage management methods)

[Slide 440] Inheritance Modes: `protected`

- `protected` inheritance: public/protected base members become protected derived members
- Derived class can be used as base class in all further derived classes
- Rarely useful
- "Controlled polymorphism": inheritance should be shared with subclasses

12.3. Polymorphic Inheritance

[Slide 441] (Non-)Polymorphic Inheritance

◯ Quiz: What is problematic about this code?

```
#include <vector>
struct Base { int a; };
struct Derived : Base { int b; Derived(int a, int b) : Base{a}, b(b) {} };
void foo(std::vector<Base>& v) {
  v.push_back(Derived(1, 2));
}
```

A. Compile error: cannot convert `Derived` to `Base`.
B. The vector only stores `Base`; the value for `b` is discarded.
C. The vector stores `Derived`, but it consumes two entries.
D. Nothing, the vector now contains a `Derived` as last element.

Solution on page 191.

[Slide 442] (Non-)Polymorphic Inheritance

△ Quiz: What is the exit code of this program?

```
struct A { int compute() { return 5; } };
struct B : public A {
    int compute() { return A::compute() + 10; }
};
int callCompute(A& a) { return a.compute(); }
int main() { B b; return callCompute(b); }
```

A. Compile error: `A::compute` – attempt to call as static member
B. Compile error: cannot pass `B` as `A&`
C. Program always exits with code 5
D. Program always exits with code 15

Solution on page 191.

[Slide 443] `virtual` Function Specifier[2]

- `virtual` enables dynamic dispatch for a function
  - ⇒ Allows function to be overridden in derived classes
  - A class with at least one virtual function is *polymorphic*
  - Overriding function can be annotated with `override` (see later)
- Calling a virtual function through pointer/reference of base class invokes behavior defined in derived class
- Suppressed when using qualified name lookup for function call

[Slide 444] `virtual`: Example

```
#include <print>
struct Base {
  virtual void foo() { std::println("Base::foo()"); }
};
struct Derived : Base {
  void foo() override { std::println("Derived::foo()"); }
};
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()
}
```

> `virtual` functions behave like methods in Java.

[Slide 445] Overriding Functions

A function overrides a virtual base class function if:

- Same name, cv-qualifiers, ref-qualifiers, and
- Same parameter type list (but not the return type)

If conditions met:

- Function is also virtual and can be overridden in derived classes
- Return type must be same or *covariant*
  - E.g., `virtual Base* m();` can be overriden by `Derived* m();`

Otherwise: function might **hide base class function**

[Slide 446] Overriding Functions: Example

[2] `https://en.cppreference.com/w/cpp/language/virtual`

```
struct Base {
  virtual void bar();
  virtual void foo();
};
struct Derived : public Base {
  void bar(); // Overrides Base::bar()
  void foo(int baz); // Hides Base::foo()
};
int main() {
  Derived d;
  Base& b = d;
  d.foo(); // ERROR: lookup finds only Derived::foo(int)
  b.foo(); // invokes Base::foo();
}
```

[Slide 447] `override` Specifier[3]

- `override`: specify that function actually overrides a virtual function
- Useful to avoid bugs where function in derived class hides base class function

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

struct Derived : public Base {
  void foo(float i) override; // ERROR: no override, different parameter types
  void bar() const override; // ERROR: no override, different cv-qualifier
};
```

> It is strongly recommendable to use the `override` specifier whenever overriding of virtual functions is intended.

[Slide 448] Final Overrider

- Final overrider: function that gets executed on virtual call
- Typically the overrider in the most derived class
- Can be more complex with multiple inheritance

Exception:

- During construction/destruction: behaves as if no more-derived classes exist
  - While constructing the base class, the derived class doesn't yet exist

⇝ Care must be taken when using virtual functions in these cases

[Slide 449] `final` Specifier

- `final` functions: cannot be overridden
- `final` classes: cannot be inherited from

[3] `https://en.cppreference.com/w/cpp/language/override`

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

[Slide 450] Destructors and Inheritance

△ **Quiz: What is the output of this program?**

```
#include <memory>
#include <print>
struct A { ~A() { std::print("A"); } };
struct B : public A { ~B() { std::print("B"); } };
int main() {
  B b;
  std::unique_ptr<A> a = std::make_unique<B>();
}
```

A. Compile error: cannot assign `unique_ptr<B>` to `unique_ptr<A>`
B. `ABA`
C. `BAA`
D. `ABAB`
E. `BABA`

Solution on page 191.

[Slide 451] Destructors and Inheritance

- Derived objects can be deleted through pointer to base class
- **Undefined behavior** unless destructor is virtual

⇒ Destructor in base class should be public and virtual; or: should be protected and non-virtual; or: you know what you are doing

```
#include <memory>
#include <print>
struct A { virtual ~A() {} };
struct B : public A { };
int main() {
  A* a = new B();
  delete a; // OK
}
```

[Slide 452] Abstract Classes[4]

- Class which cannot be instantiated, but used as a base class
- Any class with a *pure virtual* function is abstract

[4] `https://en.cppreference.com/w/cpp/language/abstract_class`

- Pure virtual function: virtual declaration ending with `= 0;`
- Pure virtual function can still be defined out-of-line

```
struct Base {
  virtual void foo() = 0; // pure virtual
};
struct Derived : Base {
  void foo() override;
};
int main() {
  Base b; // ERROR: Base is abstract
  Derived d; // OK
  Base& dr = d; // OK: pointers/references/smart pointers/etc. to abstract class
  dr.foo(); // calls Derived::foo()
}
```

[Slide 453] Pure Virtual Destructor

- Destructor can be marked as pure virtual
- Useful when class shall be abstract, but no suitable functions exists
- Out-of-line definition *must* be provided

```
struct Base {
  virtual ~Base() = 0;
};
Base::~Base() {}
int main() {
  Base b; // ERROR: Base is abstract
}
```

[Slide 454] Calling Pure Virtual Functions

△ Quiz: What is the problem with this code?

```
struct A {
  virtual ~A() { cleanup(); }
  virtual void cleanup() = 0;
};
struct B : A {
  void cleanup() override {}
};
int main() { B b; }
```

A. Compile error: cannot call pure virtual method in base class
B. Undefined behavior: calling pure virtual function in constructor/destructor
C. Semantic problem: `B::cleanup` doesn't get called, instead nothing happens
D. No problem: `B::cleanup()` gets called

Solution on page 191.

[Slide 455] Virtual Base Classes

- `virtual` base class: contained only once in the derived class, even if it occurs multiple times in the inheritance DAG
- Changes rules for unqualified name lookup
- Advice: try to avoid multiple inheritance

```
struct A {int a;};
struct B1 : virtual A {};
struct B2 : virtual A {};
struct C : B1, B2 {};
int getA(C& c) { return c.a; /* OK, only one a in C */ }
```

12.4. Type Conversions

[Slide 456] `dynamic_cast`[5]

- Convert pointers/references to classes in inheritance hierarchy
- Syntax: `dynamic_cast<`*`new-type`*`>(`*`expression`*`)`
  - *`new-type`* can be pointer or reference to class type
- Most common use case: checked/safe downcast
- Runtime check whether *`new-type`* is actually a base of the type of expression
- Failure: `nullptr` (pointers)/exception (references)
- Requires runtime type information (enabled by default)
- Other use cases: see reference

[Slide 457] `dynamic_cast`: Example

```
struct A {
  virtual ~A() = default;
};
struct B : A {
  void foo() const;
};
struct C : A {
  void bar() const;
};
void baz(const A* aptr) {
  if (const B* bptr = dynamic_cast<const B*>(aptr)) {
    bptr->foo();
  } else if (const C* cptr = dynamic_cast<const C*>(aptr)) {
    cptr->bar();
  }
}
```

[5] https://en.cppreference.com/w/cpp/language/dynamic_cast

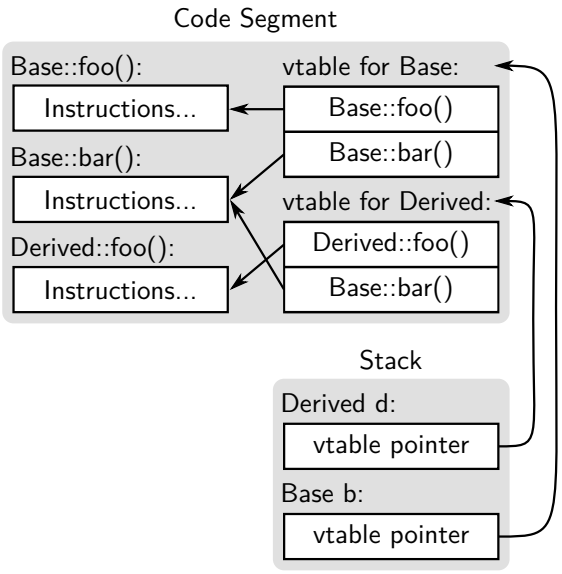


Figure 12.1.: Conceptual visualization of vtables.

12.5. Implementation of Polymorphism

[Slide 458] Implementation of Virtual Functions

- Vtable: table of function pointers to final overrider for every class
- Vtable pointer stored at beginning of every object
- Function invocation: load vtable, load fn pointer, do indirect call

```
struct Base {
  virtual void foo();
  virtual void bar();
};
struct Derived : Base {
  void foo() override;
};
int main() { Base b; Derived d; }
```

This has a very important implication: **virtual function calls are expensive!** Compilers sometimes do some optimization and speculate on the actual virtual function call (*devirtualization*), but as the compiler does not know the exhaustive set of subtypes, optimization possibilities are limited. Therefore, the two levels of indirection generally remain and cannot be eliminated.

There is also no dynamic optimization on actually occurring instance types. Managed languages like Java only support virtual functions, but at runtime track which functions are actually called and dynamically optimize and recompile code for these cases. Therefore, the overhead of dynamic dispatch in C++ is *higher* than in, e.g., Java.

The vtable layout can become rather complex with multiple inheritance; the layout

> is specified in the ABI (e.g., the Itanium C++ ABI).

[Slide 459] Implementation of `dynamic_cast`

- Vtable contains pointer to data structure that describes type
- Type checks tend to be rather expensive ⇒ noticable performance impact
- Alternative: type enum and `static_cast`

```
struct Base {
  enum class Type { Base, Derived, };
  Type type;

  Base() : type(Type::Base) {}
  Base(Type type) : type(type) {}

  virtual ~Base();
};
struct Derived : Base {
  Derived() : Base(Type::Derived) {}
};
void foo(Base* b) {
  switch (b->type) {
  case Base::Type:
    // use Base
    break;
  case Base::Derived: {
    auto* d = static_cast<Derived*>(b);
    // use Derived
    break;
  }
  }
}
```

[Slide 460] Polymorphism: Recommendations

- If performance doesn't matter: whatever
- Generally avoid `dynamic_cast`, use type enum and static cast
  - Runtime type information (RTTI) is big, cast is much more expensive
- Avoid virtual function calls where performance matters
  - Indirection is very expensive, can be very noticeable when invoked frequently
  - When important it is often possible and recommendable to avoid these

12.6. Compile-Time Polymorphism

[Slide 461] Compile-Time Polymorphism

- How to avoid virtual function calls? Templates
- Curiously Recurring Template Pattern (CRTP):[6] base class takes derived class as template parameter

[6] `https://en.cppreference.com/w/cpp/language/crtp`

```
template <class Derived> struct Base {
  Derived* derived() { return static_cast<Derived*>(this); }
  int foo() { return derived()->bar(); }
  int bar() { return 12; }

protected:
  Base() = default; // prohibit creation of Base objects
};
struct MyImpl : public Base<MyImpl> {
  int bar() { return 42; }
};
int main() { return MyImpl().foo(); } // returns 42
```

> CRTP is often used in practice to have a limited degree of polymorphism without the cost of virtual function calls.

[Slide 462] Deducing `this`

- C++23 introduces explicitly object member functions
- Type of `this` specified explicitly; `this` unusable in function body

```
struct Base {
  int foo(this auto&& self) { return self.bar(); }
  int bar() { return 12; }

protected:
  Base() = default; // prohibit creation of Base objects
};
struct MyImpl : public Base {
  int bar() { return 42; }
};
int main() { return MyImpl().foo(); } // returns 42
```

> This technique allows to write the base class of CRTP as non-template.

[Slide 463] CRTP Compared to Virtual Functions

+ No runtime overhead of virtual function calls
+ Base class can call into functions of derived class
– Definitions (generally) need to go into header files
– Less flexibility
– Cannot have container of polymorphic objects
  – I.e., no `std::vector<std::unique_ptr<Base>>`

[Slide 464] Mixins

- Mixin: compose functionality from multiple classes

> C++ doesn't natively support mixins. However, such functionality can be modeled using CRTP and multiple inheritance.

```
template <class D> struct Greeter {
  D* derived() { return static_cast<D*>(this); }
  void greet() {
    std::println("Hello, I'm {}!", derived()->name());
  }
};
struct Person : Greeter<Person> {
  std::string_view n;
  std::string_view name() const { return n; }
};
int main() {
  Person p{.n = "foo"};
  p.greet();
}
```

[Slide 465] Inheritance – Summary

- C++ supports very flexible inheritance of classes
- Classes can have zero, one, or more base classes
- Base classes can be inherited `public`/`protected`/`private`
- By default, inheritance is non-polymorphic
- `virtual` functions enable overriding and dynamic polymorphism
- Polymorphism needs care for implementing constructors/destructors
- `dynamic_cast` allows dynamic type checking and casting
- Templates allow implementing static polymorphism at compile-time
- Dynamic polymorphism often has considerable runtime overhead

[Slide 466] Inheritance – Questions

- In which order are constructors/destructors of base classes executed?
- How does inheritance change the object representation of classes?
- What is an advantage of non-polymorphic inheritance?
- What is a disadvantage of non-polymorphic inheritance?
- Why is using the `override` specifier highly recommendable?
- How are virtual functions conceptually implemented?
- Why should destructors of base classes often be virtual?
- How to use CRTP for compile-time polymorphism?

13. I/O and Testing

[Slide 468] Systems Programming in C++

- So far: mostly covered standard C++
- Standard does not contain everything required for OS interaction
  - Efficient file I/O
  - Networking
  - Direct memory allocation from the OS
  - ...
- Such operations need a different interface to the OS

> C++ provides an interface for file I/O, but this is not very efficient and often also does not provide sufficient functionality (see later).

[Slide 469] POSIX and Linux API

- POSIX: Standard defining C-API ($\rightsquigarrow$ usable in C++) for OS interaction
- Supported on most Unix-like operating systems
- Defines several data types, functions, and constants (macros) e.g. in `unistd.h`, `fcntl.h`, `sys/*.h`
- Linux defines additional types, functions, and constants
- Documented in `man` pages, usually sections 2 and 3

13.1. File I/O – POSIX

[Slide 470] File Descriptors

- File descriptor: handle to resource managed by OS
  - Files/directories in filesystem
  - Network sockets
  - Many other kernel objects
- Usually created by a function (e.g. `open`) and closed by `close`
- In C++, the RAII pattern can be very useful

[Slide 471] Opening and Creating Files

- `int open(const char* path, int flags, mode_t mode)`
  - Argument `mode` is optional, only required when file is created

- Open file at path and return fd for that file, or -1 on error
- Flags is a bitwise combination of flags and must contain exactly one of:
  - `O_RDONLY`, `O_RDWR`, `O_WRONLY`
- Flag `O_CREAT`: create file if it doesn't exist
- Flags `O_CREAT|O_EXCL`: create file if it doesn't exist, error if it does exist
- Flag `O_TRUNC`: if file exists, truncate it (remove all content)

[Slide 472] `open` Example

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
    int fd = open("/tmp/testfile", O_WRONLY | O_CREAT, 0600);
    if (fd < 0) {
        perror("/tmp/testfile");
        return 1;
    }
    close(fd);
}
```

[Slide 473] Reading/Writing Files

- `ssize_t read(int fd, void* buf, size_t count)`
- `ssize_t write(int fd, const void* buf, size_t count)`
- Read/write *up to* `count` bytes to/from `buf`
  - Can always read/write less than `count`
- Returns number of bytes read/written, -1 indicates error
- `read` return value 0: reached end of file
- Functions might block until data can be read/written
  - ⇝ Might lead to deadlocks!

> Many operations can fail with `EINTR` (interrupted). This is the only error code, in which the *intention* is that the user restarts the I/O operation, *except* for `close()`.

[Slide 474] Error Handling

- Most functions use `errno` (`<cerrno>`[1]) for error handling
- `errno`: thread-local global variable containing an error code
- If a function return -1, `errno` is set to the error code
  - `EINVAL`: invalid argument
  - `ENOENT`: no such file or directory
  - `EACCESS`: permission denied
  - ... (see `man 3 errno`)
- Error message can be retrieved using `std::strerror()` from `<cstring>`

[1] `https://en.cppreference.com/w/cpp/header/cerrno`

[Slide 475] File Positions and Seeking

- For every file descriptor: kernel remembers position in file
- `read`/`write` start at and advance that position
- `off_t lseek(int fd, off_t offset, int whence)` get and/or set current position
- `whence == SEEK_SET`: set current position to `offset`
- `whence == SEEK_CUR`: add `offset` to current position
- `whence == SEEK_END`: set current position to end of file plus `offset`
- Return value: new position in file, or -1 to indicate an error

```
int fd = open("/etc/passwd", O_RDWR);
auto fileSize = lseek(fd, 0, SEEK_END); // move to end of file
lseek(fd, -4, SEEK_CUR); // move 4 bytes backwards
write(fd, "test", 4); // overwrite the last 4 bytes
```

[Slide 476] Reading and Writing at Specific Offset

- Current offset into file is shared between all threads
- Problematic when reading/writing in parallel
- `ssize_t pread(int fd, void* buf, size_t count, off_t off)`
- `ssize_t pwrite(int fd, const void* buf, size_t count, off_t off)`
- Read/write ad specified offset, don't modify current position
- `int ftruncate(int fd, off_t length)`
- Set size of file, if larger than previous size fill with zero bytes

[Slide 477] Metadata of Files

- `int stat(const char* path, struct stat* statbuf)`
- `int fstat(int fd, struct stat* statbuf)`
  - `<sys/types.h>`, `<sys/stat.h>`, `<unistd.h>`
- Write metadata of specified path into `statbuf`
- `st_mode`: File type and mode (e.g., permissions)
- `st_uid`: user id of the file owner
- `st_size`: size of the file
- `st_mtime`: timestamp of last modification
- ...

[Slide 478] UNIX File Types

- Regular files
- Directories
- Symbolic links (often implicitly followed)
- Pipes
- Character devices (e.g., terminal, `/dev/urandom`)
- Block devices (e.g., disks)
- Sockets

- `st_size` shows actual size *only* for regular files and block devices

[Slide 479] Checking for Non-Existing Files

△ **Quiz: What is NOT problematic about this code?**

```
// Includes <err.h> <fcntl.h> <sys/stat.h>
struct stat statbuf;
if (stat(argv[1], &statbuf) < 0)
    err(1, "%s", argv[1]);
if (!S_ISREG(statbuf.st_mode))
    errx(1, "%s: Not a regular file", argv[1]);
int fd = open(argv[1], O_RDONLY); // No need to check error, file exists
```

A. Process might not have permission to open/read the file
B. `stat` doesn't follow symbolic links, but `open` does
C. `stat` might refer to a different file than `open`
D. `open` might return `EINTR`, where the function should be restarted

13.2. C++ Streams

[Slide 480] C++ Streams[2]

- C++ library for I/O designed around the concepts of *streams*
- `std::istream`: base class for input operations (`operator>>`)
- `std::ostream`: base class for output operations (`operator<<`)
- `std::iostream`: subclass of `std::istream` and `std::ostream`
- `std::cin`/`std::cout`: streams for standard input/output
- Like `std::string`, actually templates parameterized for `char`

[Slide 481] Input and Output Streams

- `operator>>()`: read value of given type, skip leading whitespace
- `operator<<()`: write value of given type
  - Both operators can be overloaded for own types as second argument
- `get()`/`put()`: read/write single character
- `read()`/`write()`: read/write multiple characters

```
// Defined by the standard library:
std::istream& operator>>(std::istream&, int&);
int value;
std::cin >> value;

// Write 1024 chars to cout:
std::vector<char> buffer(1024);
std::cout.write(buffer.data(), 1024);
```

[2] https://en.cppreference.com/w/cpp/io

[Slide 482] Common Operations

- Various methods to check whether stream is in specific error state
- `good()`: no error occurred
- `fail()`: an error occurred
- `bad()`: a non-recoverable error occurred
- `eof()`: reached end-of-file
- `operator bool()`: true if stream has no errors

```
int value;
if (std::cin >> value) {
    std::cout << "value␣=␣" << value << std::endl;
} else {
    std::cout << "error" << std::endl;
}
```

[Slide 483] `std::endl`

△ **Quiz: Which statement is correct?**

A. `std::cout << std::endl` is equivalent to `std::endl(std::cout)`.
B. `std::cout << std::endl` is equivalent to `std::cout << '\n'`.
C. `std::endl` is an object type and `operator<<` has a special overload.
D. `std::endl` is more efficient than writing a new line character.

Solution on page 191.

- Flushing an output stream is often not necessary
- Prefer writing newline characters instead

[Slide 484] File Streams

- `std::ifstream`: file stream to read file
- `std::ofstream`: file stream to write file
- `std::fstream`: file stream to read an write 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";
}
```

[Slide 485] Reading a File Into Memory

```
std::string readFile(const char* path) {
  auto stream = std::ifstream(path, std::ios::in);
  stream.seekg(0, std::ios::end);
  auto size = stream.tellg();
  stream.seekg(0, std::ios::beg);
  std::vector<char> data(size);
  stream.read(&data[0], size);
  return std::string(&data[0], size);
}
```

- This is *not* how to do it

> This simple function is broken in almost every possible regard:
>
> - Errors are ignored.
> - `tellg` doesn't report the size of the file, but merely indicates a position into the file. This happens to be the same as the file size on some systems, but not on others (Windows).
> - Some files don't have a size or don't support seeking, e.g., character devices (terminals) and pipes. In such cases, there is no way to determine the size.
>
> A general possibility is to read chunks of the file until the end of file is reached.
>
> ```
> // ...
> std::string res;
> std::vector<char> buf(0x2000);
> do {
>     stream.read(&buf[0], buf.size());
>     res.append(&buf[0], 0, stream.gcount());
> } while (stream.good());
> // ...
> ```

[Slide 486] Disadvantages of Streams

- Streams make heavy use of virtual functions and virtual inheritance
- System's locale settings are respected ⇝ slower
  - E.g., whether dot or comma is used for floating-point numbers
  - Especially handling of numbers is very inefficient
- Streams have implicit state (e.g., formatting specifiers, error status)
- Many important operations (e.g. `stat`) are not exposed, no way of accessing the underlying file descriptor

⇒ Avoid using C++ streams, better use OS-specific functions

[Slide 487] I/O Performance and Buffering

- I/O operations are often slow (e.g., hard disk, network, etc.)
  - ⇒ Kernel doesn't immediately write file to disk
  - Instead, writing data is often delayed for some time
  - Buffers flushed on close or `fsync`
- System calls are somewhat slow (context switch, etc.)

  - ⇒ Standard library doesn't immediately calls kernel
  - Instead, data is buffered in user-space for some time
  - Buffers flushed on close, exit, or flush
- Techniques for more efficient I/O: `mmap`, `io_uring`, ... all of these are somewhat-to-very OS-specific and non-portable.

[Slide 488] `close`

△ **Quiz: What can happen when the error of `close()` is ignored?**

A. Silent data loss.
B. File descriptor leak.
C. Nothing, `close` cannot return an error.

Solution on page 192.

13.3. C++ Filesystem Library

[Slide 489] std::filesystem[3]

- C++17 addition, provides interface for working with paths and files
- Provides abstractions for several POSIX functions
  - But: not all, and often doesn't expose the required interface
- `std::filesystem::path` is useful for working with file paths
  - Convenience functions for concatenating, adding suffixes, etc.
- Cannot provide the same guarantees as OS-defined functions

13.4. Testing

[Slide 490] Testing

Tests should be an integral part of every larger project

- Unit tests
- Integration tests
- ...

Good test coverage greatly facilitates implementing a large project

- Tests can ensure (to some extent) that modifications do not break existing functionality
- Can easily refactor code
- Can easily change the internals of a component
- ...

[3] `https://en.cppreference.com/w/cpp/filesystem`

[Slide 491] Googletest (1)

- Works on a large variety of platforms
- Contains a large set of useful functions
- Can usually be installed through a package manager
- Can be added to a CMake project through the `FindGTest.cmake` module

Functionality overview

- Test cases
- Predefined and user-defined assertions
- Death tests
- ...

[Slide 492] Googletest (2)

Simple tests

```
#include <gtest/gtest.h>
TEST(TestSuiteName, TestName) {
    ...
}
```

- Defines and names a test function that belongs to a test suite
- Test suites can for example map to one class or function
- Googletest assertions can be used to control the outcome of the test function
- If any assertion fails or the test function crashes, the entire test case fails

[Slide 493] Googletest (3)

Fatal assertions

- Fatal assertions are prefixed with `ASSERT_`
- When a fatal assertion fails the test function is immediately terminated

Non-fatal assertions

- Non-fatal assertions are prefixed with `EXPECT_`
- When a non-fatal assertion fails the test function is allowed to continue
- Nevertheless the test case will fail
- All assertions exist in fatal and non-fatal versions

Assertion examples

- `ASSERT_TRUE(condition);` or `ASSERT_FALSE(condition);`
- `ASSERT_EQ(val1, val2);` or `ASSERT_NE(val1, val2);`
- ...

[Slide 494] Googletest (4)

A custom `main` function needs to be provided for Googletest

```
#include <gtest/gtest.h>
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- Should usually be placed in a separate `Tester.cpp` or `main.cpp`

[Slide 495] Example: Average of Two Integers

- Compute the average of two integers, round toward the first

(see script)

A simple, naive implementation of the function `average`:

```
#include <concepts>

template <std::integral T>
T average(T a, T b) {
    return (a + b) / 2;
}
```

Now we write some tests and compile these with `-fsanitize=undefined`:

```
#include <gtest/gtest.h>
#include "avg.hpp"

TEST(avg, average) {
    EXPECT_EQ(average<int>(1, 3), 2);
    EXPECT_EQ(average<int>(-1, -3), -2);
    EXPECT_EQ(average<int>(0x7ffffffe, 2), int{0x40000000});
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Oh, no! We get an integer overflow:

```
runtime error: signed integer overflow: 2147483646 + 2 cannot be represented in type 'int'
```

So we come up with an alternative implementation and add some more tests:

```
#include <concepts>

template <std::integral T>
T average(T a, T b) {
    return a + (b - a) / 2;
}

// Tests...
    EXPECT_EQ(average<int>(-0x7ffffffe, -2), int{-0x40000000});
    EXPECT_EQ(average<int>(0x7ffffffe, -0x7ffffffe), int{0})
```

Another integer overflow:

```
runtime error: signed integer overflow: -2147483646 - 2147483646 cannot be represented in type 'int'
```

Ok, let's use unsigned integers to avoid this undefined behavior:

```
#include <concepts>
#include <type_traits>

template <std::integral T>
T average(T a, T b) {
    using UT = std::make_unsigned_t<T>;
    return UT(a) + (UT(b) - UT(a)) / 2;
}
```

Now we again some test fails:

```
Expected equality of these values:
  average<int>(-1, -3)
    Which is: 2147483646
  -2
```

Google must know how to solve this problem, don't they?[a]

```
#include <concepts>
#include <type_traits>

template <std::integral T>
T average(T a, T b) {
    using UT = std::make_unsigned_t<T>;
    return (UT(a) + UT(b)) >> 1;
}
```

... but apparently not. Let's go back. The problem appears when `a > b`. So just swap operands?

```
#include <concepts>
#include <type_traits>

template <std::integral T>
T average(T a, T b) {
    using UT = std::make_unsigned_t<T>;
    int sign = 1;
    UT a1 = a, b1 = b;
    if (a > b) {
        sign = -1;
        a1 = b, b1 = a;
    }
    return a + sign * ((b1 - a1) / 2);
}

// More tests
    EXPECT_EQ(average<signed char>(-3, -4), -3);
    EXPECT_EQ(average<signed char>(-4, -3), -4);
    EXPECT_EQ(average<unsigned>(1, 3), 2);
    EXPECT_EQ(average<unsigned>(0xffffffff, 1), 0x80000000u);
    EXPECT_EQ(average<unsigned>(0xffffffff, 0xfffffffe), 0xffffffffu);
    EXPECT_EQ(average<unsigned>(0xfffffffe, 0xffffffff), 0xfffffffeu);
```

In any case, use `std::midpoint` instead. (NB: I don't guarantee for the correctness of this code.)

[a] https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-

[Slide 496] Coverage (1)

Code coverage can help ensure proper testing of a project

- Simple metrics like line coverage have to be interpreted carefully
- Can indicate that a certain part of a project has *not* been tested properly
- Can usually *not* indicate that a certain part of a project has been tested exhaustively

Line coverage information can automatically be collected during test execution

- Possible with a variety of tools
- GCC contains the built-in coverage tool `gcov`
- Clang can produce `gcov`-like output
- `lcov` together with `genhtml` can be used to generate HTML line coverage reports from information collected during test execution

[Slide 497] Coverage (2)

Brief example

```
# build executable with gcov enabled
> g++ -fprofile-arcs -ftest-coverage -o main main.cpp

# run executable and generate coverage data
> ./main

# generate lcov report
> lcov -c --directory . --output-file coverage.info --ignore-errors mismatch

# generate html report
> genhtml coverage.info --output-directory coverage
```

- Produces HTML coverage report in `coverage/index.html`
- Configuration for coverage reports should be part of CMake configuration

[Slide 498] Integration Tests

- Writing fine-granular unit tests can be quite tedious
- High overhead when refactoring code: need to adjust all tests
- In practice: unit tests complemented with integration tests
- For example: test I/O behavior of the entire program

> Aside from that, units do not necessarily need to correspond to C++ classes or files. A unit is typically an entire component that has a well-defined API that is unlikely to change often.

[Slide 499] `FileCheck`[4] Tests

- `FileCheck`: utility from LLVM to verify output against expectation

[4] `https://llvm.org/docs/CommandGuide/FileCheck.html`

```
// llvm-project/clang/test/Lexer/counter.c
// RUN: %clang -E %s | FileCheck %s

#define PASTE2(x,y) x##y
#define PASTE1(x,y) PASTE2(x,y)
#define UNIQUE(x) PASTE1(x,__COUNTER__)

A: __COUNTER__
B: UNIQUE(foo);
C: UNIQUE(foo);
D: __COUNTER__
// CHECK: A: 0
// CHECK: B: foo1;
// CHECK: C: foo2;
// CHECK: D: 3
```

[Slide 500] Auto-Generating Tests

- Sometimes, expected output of tests can change
- Sometimes, this is due to unrelated changes
  - E.g., when adding an optimization to a compiler, the output of other tests changes
- Adjusting all tests manually is a huge effort
- Having a tool to auto-generate the expected output reduces this
  - Only need to review code changes in `git diff`

[Slide 501] I/O and Testing – Summary

- POSIX provides a somewhat portable and rather low-level operating system interface for interacting with the file system
- File I/O in POSIX centered around file descriptors
- C++ I/O designed around streams as a higher-level abstraction
- C++ streams are inefficient and limited in features
- C++ Filesystem API provides good abstraction for paths
- Unit tests and integration tests are important for quality

[Slide 502] I/O and Testing – Questions

- When do `read`/`write` return? What does a return value 0 imply?
- What types of errors can occur during `close()`?
- How to reliably get the size of a file for reading it into memory?
- What is the difference between `bad()` and `fail()` on streams?
- What are disadvantages of streams over using OS-specific functions directly?
- How to get code coverage information from unit tests? What does this mean?
- What are benefits of integration tests over unit tests?

A. Exercise Solutions

[Slide 369]

D. `throw` is an expression, more precisely a prvalue of type `void`. During exception unwinding, the destructors of variables with automatic storage duration in the callers up to the `catch` handler will be called, in this case, the memory stored held by `ui` will be freed as part of the exception unwinding. Inside the throw expression, `x` evaluates to the type `int`, which is caught by the handler in `main`. Therefore, the program exits with status code zero, as the `return` statement in `main` can be omitted.

[Slide 371]

B. The constructor of `B` uses a *function try block*, which can also handle exceptions thrown during the initialization of members (or during the execution of destructors of the function body). For a constructor/destructor try block, the currently handled exception is rethrown at the end of a `catch` handler, so the `throw;` is not required.

No objects of type `A` can exist, because the only constructor of `A` unconditionally throws an exception. Therefore, also no objects of `B` cannot exist, because that would require the existence of an object of type `A`.

[Slide 378]

C. `std::string` is just a type alias for `std::basic_string<char>`, so a constructor/destructor with the name `string` does not exist.

`delete(s1)` is equivalent to `delete s1` and would attempt destroy an object allocated by (non-placement-)`new`. `s1` was, however, not allocated by the default `operator new`, and therefore this would be undefined behavior.

Manually destructing `ta` is not correct: the destructor gets automatically called when the variable goes out of scope at the end of `main`; explicitly calling the destructor would cause the destructor to be called twice, which is undefined behavior.

As the `std::string` pointed to by `s1` was constructed manually, it also needs to be destructed manually; the compiler cannot know at which point this should happen (similar to dynamic storage duration).

[Slide 382]

`C`. Reading an inactive union member is undefined behavior. This is unlike C, where this is merely unspecified (and results in a type-punning read where the bits are reinterpreted).

[Slide 389]

A. If a vector moves its elements while growing and one of these move operations throws an exception, some elements are in the new allocation while others are not. Such a situation is generally not recoverable, as it might not be possible to move elements back.

[Slide 400]

C. `getMsg` returns a reference to the string owned by `this`, so the lifetime of `*this` should be longer than the lifetime of the returned value.

[Slide 405]

C. The assignment to a `constexpr` variable requires the initializer to be a compile-time constant expression; but the pointer created by `new int` cannot be possibly known at compile-time.

[Slide 408]

D. `constexpr` functions are implicitly `inline`.

Non-`constexpr`/`consteval` can be evaluated at compile-time if they have no side effects, depending on compiler optimizations. `constexpr` merely permits evaluation of a function as constant expression, but this is not required. `consteval` functions can include code that is not evaluable at compile-time, if the evaluation is not attempted.

[Slide 410]

D. The heap allocation of `std::vector` never escapes to run-time (more precisely: the pointer does not escape and none of the restrictions for compile-time evaluation are violated).

[Slide 416]

C. Due to integer promotion, before the evaluation of the multiplication in `mul16`, `a` and `b` get promoted to `signed int`, as the type of `uint16_t` (typically `unsigned short`) has a lower rank. The result of `0xffff * 0xffff` is `0xfffe0001`, which cannot be represented in a signed 32-bit integer — this is a signed integer overflow, which is undefined behavior. As undefined behavior cannot occur in a constant expression, the result is not a constant expression and therefore cannot be used in a `static_assert`.

[Slide 421]

C. `node_type` is not accessible outside of `MyGraph` and therefore the static assertion fails.

[Slide 432]

D. The user-defined copy constructor of `B` does not delegate to the copy constructor of `A`, but to its default constructor. The proper implementation would be:

```
struct B : A {
    B() { std::println("B"); }
    B(const B& other) : A(other) { std::println("b"); }
};
```

[Slide 433]

E. Base classes are constructed first and in left-to-right order and therefore destructed last and in right-to-left order.

[Slide 434]

C. For `D`, first `B` is constructed, then `C`, then `D` itself. For `B` (and similarly, `C`), first `A` is constructed and then `B` itself. This implies that there are *two* objects of type `A` in `D`!

[Slide 441]

B. The vector only stores elements of type `Base`. A `Derived&&` is implicitly convertible to a `Base&&`, thereby invoking the move constructor of `Base` in call to `push_back`.

[Slide 442]

C. Inheritance is non-polymorphic unless specified otherwise. `callCompute` only operates on objects of type `A` and therefore calls the method `compute` of the `struct A`.

[Slide 450]

B. This quiz shows a very dangerous pattern: the destructor of `A` is not virtual and therefore the deleter of the `unique_ptr<A>` will not call the destructor of `B`! Variables with automatic storage duration are destructed in reverse order, so the `unique_ptr` gets destroyed (`A`) before `b` (`BA`).

The assignment works, this is essentially an assignment of the form `A* a = new B()`, which is permitted.

[Slide 454]

B. In a constructor/destructor, virtual function calls behave as if no more derived class exists. Calling a pure virtual function is undefined behavior.

[Slide 483]

A. `basic_ostream::operator<<` has an overload that takes a function pointer as second parameter, which simply calls the function with the stream as parameter. `std::endl` is a function (template) which writes a newline character and flushes the stream.

[Slide 488]

A. There might be an I/O error while flushing the kernel buffers.