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!

```cpp
#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]

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

[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

[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 ⇝ 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)

<table>
<tr><th>Prec.</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>11</td><td><code>&</code></td><td>Bitwise AND</td><td>left-to-right</td></tr>
<tr><td>12</td><td><code>^</code></td><td>Bitwise XOR</td><td>left-to-right</td></tr>
<tr><td>13</td><td><code>|</code></td><td>Bitwise OR</td><td>left-to-right</td></tr>
<tr><td>14</td><td><code>&&</code></td><td>Logical AND</td><td>left-to-right</td></tr>
<tr><td>15</td><td><code>||</code></td><td>Logical OR</td><td>left-to-right</td></tr>
<tr><td>16</td><td><code>a?b:c</code>
<code>throw</code>
<code>=</code>
<code>+= -= *= /= %=</code>
<code><<= >>= &= ^= |=</code></td><td>Ternary conditional
throw operator
Direct assignment
Compound assignment
Compound assignment</td><td>right-to-left</td></tr>
<tr><td>17</td><td><code>,</code></td><td>Comma</td><td>left-to-right</td></tr>
</table>

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 ⇝ **anything** can happen
  - Typically cases, where checks would be costly or impossible

⇒ 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;
  unsigned meaningOfLife = 42;
}
```

[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

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

[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

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

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

```
  // 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]

```
for (unsigned i = 0; i < 10; ++i) {
  // iterate 0, 1, 2, ..., 9
}
for (unsigned i = 0, len = getLength(); i != len; ++i) {
```

[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

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

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

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

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

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

- 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)

[23] `https://en.cppreference.com/w/cpp/language/object`

  - 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

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

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

[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.
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.

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

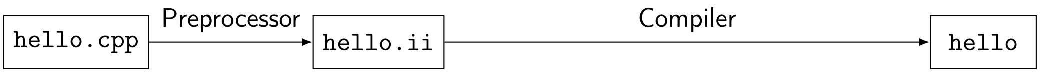
[Slide 65] On “Internet”

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

Quiz: What does the function f return?

[3] `https://en.cppreference.com/w/cpp/preprocessor/replace`
[4] NB: Re-defining keywords is undefined behavior if the standard library is included.

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

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

[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

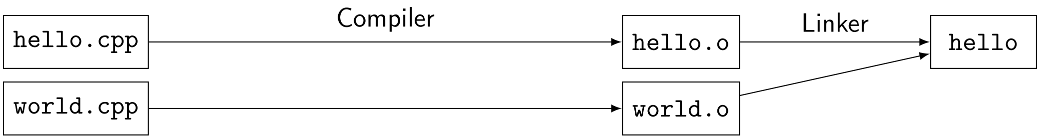
```
void function() {
  int a = 4;
  const int b = a;
  a = 0; // OK
  b = 10; // ERROR: assignment of read-only variable
  volatile int v = 5; // will not be optimized out
}
```

[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

```
//--- 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 `Makefiles`, 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

⇝ 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

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

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

```
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;

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

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

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

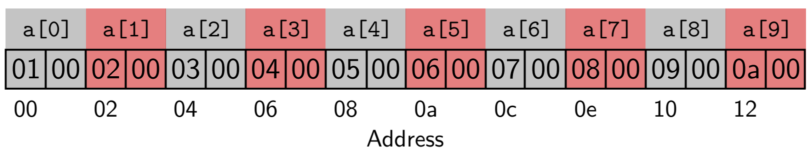
[Slide 121] Array Initialization

- 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
}
```

[Slide 127] `std::array`

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

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

```
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]

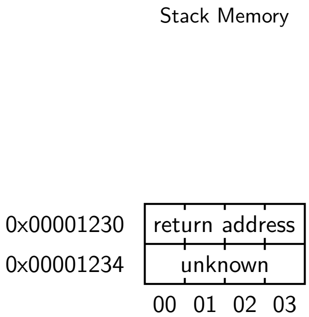
- Operator `*`: obtain lvalue reference to pointed-to object
- Operand must be a pointer, cv-qualification is retained
- Also referred to as *pointer dereference*

```
int a = 10;
int* ap = &a;
int& ar = *ap;
ar = 20; // a is now 20
*ap = 4; // a is now 4
```

[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

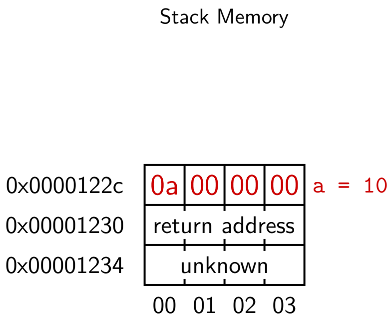
[Slide 134] What is Happening? (1)

```
int main() {

}
```



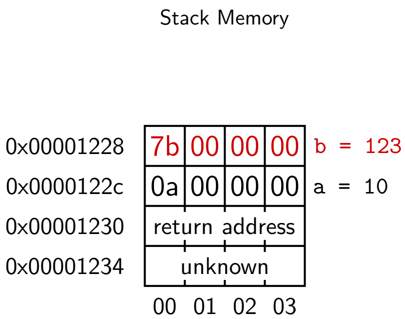
[Slide 135] What is Happening? (2)

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

}
```



[Slide 136] What is Happening? (3)

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

}
```



[Slide 137] What is Happening? (4)

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



}
```

Stack Memory

	00	01	02	03	
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	0a	00	00	00	a = 10
0x00001230	return address				
0x00001234	unknown				

[Slide 138] What is Happening? (5)

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


}
```

Stack Memory

	00	01	02	03	
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	2a	00	00	00	a = 42
0x00001230	return address				
0x00001234	unknown				

[Slide 139] What is Happening? (6)

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

}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	2a	00	00	00	a = 42
0x00001230	return address				
0x00001234	unknown				

[Slide 140] What is Happening? (7)

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




}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				

[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;



}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	28	12	00	00	c = 0x1228
0x00001228	7b	00	00	00	b = 123
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				

[Slide 142] What is Happening? (9)

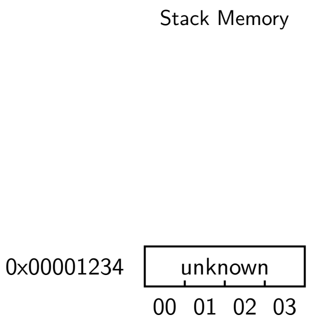
```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;


}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	28	12	00	00	c = 0x1228
0x00001228	18	00	00	00	b = 24
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				

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

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

> 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);

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

[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

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

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

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

[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

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

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

[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
    // (2 padding bytes)
    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
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*

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

- ... 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
- 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};`

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

[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)
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);
```

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

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
    }
};
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 `friends` of their parent

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

```
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

[8] `https://en.cppreference.com/w/cpp/language/types#Incomplete_type`

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-();
```

[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);
```

[9] https://en.cppreference.com/w/cpp/language/operators
[10] https://en.cppreference.com/w/cpp/language/operator_arithmetic
[11] https://en.cppreference.com/w/cpp/language/operator_comparison

[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

- `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* `[]`

[12] https://en.cppreference.com/w/cpp/utility/compare/partial_ordering
[13] https://en.cppreference.com/w/cpp/language/operator_incdec
[14] https://en.cppreference.com/w/cpp/language/operator_member_access

- `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`

```
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 {
```

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

```
        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
```

[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
}
```

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

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

```
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 {
```

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

```
    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?
- 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?