C++ Introduction

C++ Introduction

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad (🦭)

Slides:

https://github.com/PeterSommerlad/talks_public/tree/master/

My philosophy

Less Code

=

More Software

Speaker notes

I borrowed this philosophy from Kevlin Henney.

In this course

- Modern C++17 (with a glimpse of C++20)
- IDE Cevelop
- C++ Unit Testing Easier library (CUTE)

Speaker notes

While you might prefer other IDEs, I chose the Eclipse-CDT-based IDE Cevelop that my former team at IFS Institute for Software created.

Cevelop provides some checkers for typical beginner mistakas as well as good support for writing Unit Tests with my test framework CUTE (https::/cute-test.com). The latter is important, because CUTE relies on the IDE to automatically generate test registration code.

Not in this course

- *All* of C++
- Building C++ on the command line
- C++ build systems (cmake, scons, make)
- C++ package manager (conan, vcpkg)
- other C++ Unit Test Frameworks (Catch2, GoogleTest)
- C++20, C++98
- Other C++ IDEs (vscode, clion)

Speaker notes

You can observe the command line used by Cevelop in its Console window.

We will neither look at all features of C++20 nor of the limitations of previous C++ language standards.

C++ Resources

- ISO C++ standardization
- C++ Reference
- Compiler Explorer
- C++ Core Guidelines
- Hacking C++ reference sheets
- Our Exercises

Speaker notes

complete link texts, in case hyperlinks vanish from PDF

- https://isocpp.org/
- https://en.cppreference.com/w/
- https://compiler-explorer.com/
- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- https://hackingcpp.com/
- https://github.com/PeterSommerlad/CPPCourseIntroduction

C++ Genealogy

C++98
initial standardized version

C++03
bug-fix of C++98, no new features

C++11
major release (known as C++0x): lambdas, constexpr, threads, variadic templates

C++14
fixes and extends C++11 features: variable templates, generic lambdas

C++17
(almost) completes C++11 features: CTAD, better lambdas

C++20
new major extension: concepts, coroutines, modules, constexpr “heap”

C++23
feature-complete (2022-02), fixes/extends C++20

Speaker notes

The ISO standardization process now uses a three year release cycle. However, for major releases it takes time for implementors to provide the new language features and library. Most C++ compilers do not yet have fully implemented C++20 and some implementation diverge in subtle details, because the specification is inaccurate. This is a typical chicken-egg problem: * compilers will only implement language features in production quality, when they are part of the standard * specification in the standard is only scrutinized when independent compiler/library authors implement them

C++20 modules and coroutines are not yet generally usable across compilers. Concepts are.

A minimal valid C++ program

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


Speaker notes

In C++ we code the functionality of our program within functions.

Each function has a

- return type (`int`)
- name (`main`) - where `main` is special
- pair of parentheses (`()`) enclosing parameters - even if none is defined
- body (`{}`(,cpp)) - statements enclosed in curly braces

The `main()` function is special, because

- it is the only one, where the return statement can be omitted,
- it defines the code that makes the whole program, and
- it is the only one that cannot be called from, therefore,
- it cannot be tested from within C++ code.

Hello world

```
//============================================================================

// Name        : hello.cpp
// Author      : Peter Sommerlad
// Version     : 0.0
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================


#include <iostream>
using namespace std;

int main() {
        cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
        return 0;
}
```


Speaker notes

What is wrong with this hello world program?

Less Code

=

More Software

Kevlin Henney and Peter Sommerlad

Speaker notes

Remember my philosophy!

Hello world simplified

```
#include <iostream>

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

What is still bad with this code?

Speaker notes

The essence of the code is not easily testable, only program observation will tell its behavior

While obvious in this case, it is easily non-obvious in normal code

Also dependency to global variable makes it untestable

Producing output

```
        out << "!!!Hello World!!!\n";
```

- Output is via `std::ostream` objects and uses the `<<` operator.
- `<<` can be chained to output multiple values:

```
  out << "Hello " << name << ", how are you?\n";
```


Speaker notes

Overloading operators is an essential features of C++ that we will look at. The order of the arguments is significant. For now, the built-in meaning of `<<` is “left bit shift” but for stream objects it means output.

Hello world alternative

```cpp
#include <fmt/core.h>

int main() {
        fmt::print("Hello {}!\n", "Cpp");
}
```

the fmt library provides a more modern alternative for output

Speaker notes

C++20 provides the format library as part of its standard via

```cpp
#include <format>
```

However, today, the format library provides a better alternative even for earlier C++ versions.

see https://github.com/fmtlib/fmt

Testable Code

- no hard-coded dependencies to globals (`std::cout`)
  - except for `constexpr` compile-time constants
- pass globals down the call chain from `main()`
- unit-test your functions
  - test also for error conditions and handling
  - test the good cases
  - but only test what can go wrong
  - always test for what went wrong once

Speaker notes

While my explanations might omit corresponding tests, all code that we write should be accompanied by automated test.

Starting with a test case, makes it easier to understand what we want to achieve with a function and as a side effect tends to reduce dependencies.

We will see and exercise how to create a testable hello world program after we looked at how C++ compilation works.

C++ compilation model

C++ uses a “separate” compilation model.

Multiple *Translation Units* (TU) form a program.

Things defined in one TU are made visible in other TUs through declarations.

Speaker notes

The compilation model of C++ was inherited from C. This allows independent working on different modules, while combining large programs later.

Declaration
defines the existence of a “named” thing

Definition
defines a “named” thing

The “things” that can be defined in C++ are variables, functions, types, or templates. C++20 adds concepts. In addition namespaces have a name, but are not introduced via “declarations”.

C++ source files

- *.cpp files for source code
  - “implementation file”
  - function implementations (unless inline)
  - translation unit - compiler argument
- *.h (or* .hpp) files for interfaces
  - “header file”
  - declarations and definitions for use
  - textual inclusion
  - `#include "header.h"`

Speaker notes

Header files that belong to the C++ standard library lack the “.h” extension and use angle brackets for the inclusion:

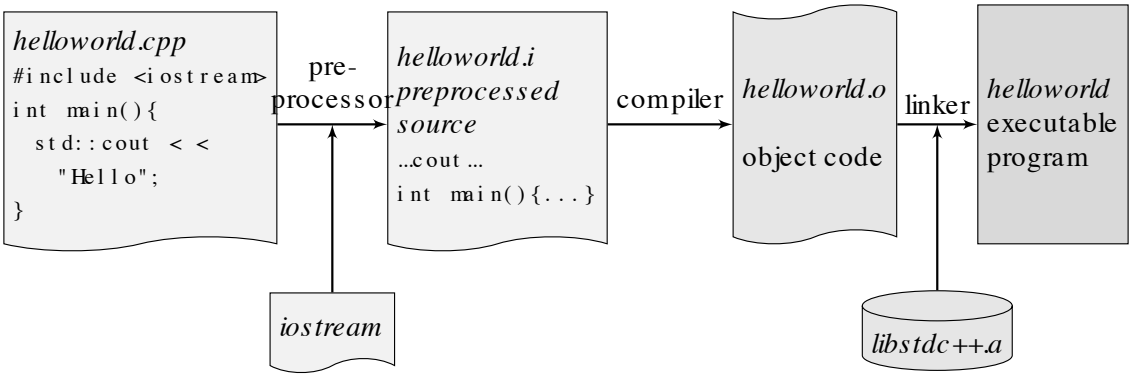
```
#include <string>
```

Function definitions in header files are for inline functions:

- **`constexpr`** functions
- class member functions within class
- function templates
- other functions explicitly marked with **`inline`**

`inline` does not guarantee “inlining” of the function body. Not providing “inline” does also not prevent the compiler from inlining the function body instead of a call (at least, when optimisation is not turned off)

Compiling C++



Speaker notes

Three phases of compilation:

1. preprocessor - textual replacement of # directives (and preprocessor macros)
2. compiler - translation of C++ into machine/object code (source to object file)
3. linker - combine object code and libraries to executable program

C++ 20 provides a new **`module`** mechanism, that is unfortunately not yet universally usable across all compilers. The **`import`** of C++20 modules will reduce the need for textual inclusion of header files, one will import another module instead. However, the details of the C++20 specification are not consistently implemented across those compiler that already provide the mechanism. Therefore, we will refrain from introducing the module mechanism here. It will only become usable well with C++23, that will provide the content of the standard library via a **`import module`** `std;`

Untestable Hello world

```cpp
#include <iostream>

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

What is still bad with this code?

We cannot test anything in ***`main()`***

Speaker notes

The essence of the code is not easily testable, only program observation will tell its behavior

While obvious in this case, it is easily non-obvious in normal code

Also dependency to global variable makes it untestable

Testing Output

use a function parameter ***`std::ostream &out`***

```
void sayhello(std::ostream &out) {
        out << "!!!Hello World!!!\n";
}
```

`&` *in front of the parameter means* ***pass by reference***.
This is used for ***side effects***.

Speaker notes

We cannot test output using `std::cout`. However, `std::cout` is a specialized version of `std::ostream`. The type `std::ostringstream` can be used in tests to collect output written to an `ostream`.

Use *reference parameters* only, when there must be a side effect on the function call argument. When a function has a (non-const lvalue) reference parameter, the argument at the call site must be a variable (mutable lvalue). Since output is a side effect on a stream object and because we cannot pass a stream object to a function otherwise, it is OK. Normal Parameters of the form `type name` are “pass by value”.

The output created by the output operator `<<` depends on the type of the variable. There is no need for a matching formatting character as in C, for example.

Hello world: extract function

hellomain.cpp

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

int main() {
        sayhello(std::cout);
}
```

hellotest.cpp

```
#include "sayhello.h"

void testSayHello() {
  std::ostringstream out{};
  sayhello(out);
  ASSERT_EQUAL("Hello World\n",
    out.str());
}
```

sayhello.h

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
#include <iosfwd>
void sayhello(std::ostream &out);
#endif /* SAYHELLO_H_ */
```

sayhello.cpp

```
#include "sayhello.h"
#include <ostream>
void sayhello(std::ostream &out) {
        out << "!!!Hello World!!!\n";
}
```

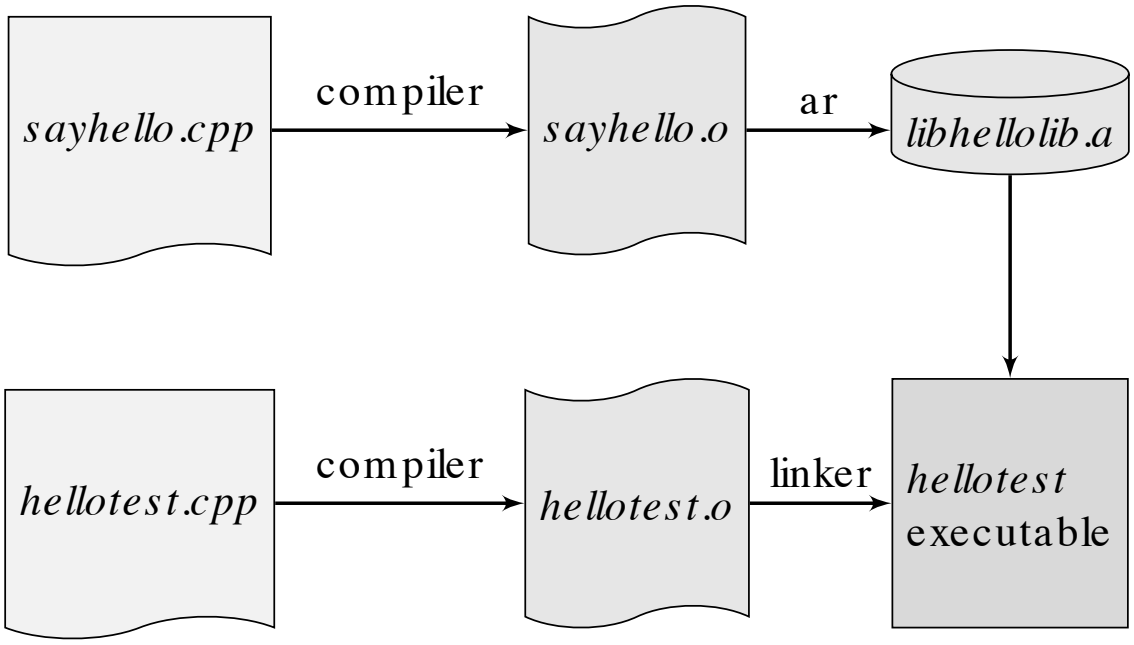

Speaker notes

here we have separated the functionality from main into a function that we put into a separate translation unit and library. This way, our hello world program becomes testable.

Note, that the global variable is only used within main.

This parameter passing is sometimes referred to as “dependency injection” to confuse people

Compiling C++ for tests



Speaker notes

For compiling tests, we put the functions to be tested in a separate library (create a library project).

The object files of the functions are combined to a library with the archiver program `ar`. Using ar will result in a so called static library.

For shared objects/dynamic link libraries a similar mechanism is used. Those libraries can change independently of the executable program and are combined with the executable when the program starts.

Then the tests against those functions will be put into a test project, that refers to the library project.

For compiling the test project the following command line information must be present:

```
g++ -I<path-to-library-header-files> -L<path-to-library> -l<nameoflibrarywithoutlib>
```

```
g++ -I../hellolib -L../hellolib/Debug -lhellolib
```

An IDE or a build system should set all things automatically.

Getting Input

```
std::istream >> variable
```

sayhello.cpp

```
std::string inputName(std::istream &in){
  std::string name{};
  in >> name;
  return name;
}
```

The type of the variable determines what is input.

Speaker notes

Input from a stream can fail for various reasons, for example,

- There is no more input (`.eof()`)
- The input does not match the variable type (`.fail()`)
- The input file is closed or otherwise corrupt (`.bad()`()

If everything is OK after input the stream is `.good()`

For a `std::string` the input operator `>>` skips blanks and newlines (aka whitespace) and reads characters up to the next whitespace character or up to the end of the stream. When the variable to be read is of numerical type, like `int`, the input operator `>>` expects digits to follow any optional whitespace characters.

Exercise 1

exercise01

Speaker notes

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise01/

C++ Terminology

Speaker notes

Here we introduce important C++ concepts briefly to establish terminology.

Important Terms

Term	Meaning	Example
Value	element of a Type	`42`
Type	range of Values and behavior	`int, bool`
Variable	named holder of a Value	`int const x{42};`
Expression	computes a Value of a Type	`6 * 7 == x`
Statement	computational step	`break;`
Declaration	introduce a Name	
Definition	specify a Name	
Function	combines statements	`void f(){}`

Speaker notes

just a brief overview

Names

- A name abstracts a C++ element.
- Each name used, must first be introduced.
- Except for some built-in names.
  - e.g., `int double float`
- Library names are introduced with `#include`

*You introduce a name with a **Declaration***

Speaker notes

C++ requires each name to be introduced first, before it can be used.

A **Name** is introduced through a **Declaration**

Names are the key to abstraction, by giving a name to something, it becomes usable in other places.

For example, a variable provides a name for a value. Using the variable name (multiple times) refers to the value, without repeating the computation or literal that created the value.

Built-in types

their names are keywords and always available

- `bool`: **`true false`**
- `int`: signed integers
- `unsigned`: unsigned integers
- `long long`: signed integers with greater range
- `float`: floating point numbers
- `double`: floating point with more precision and range

Types define a range of values and the possible operations using those values

Speaker notes

Most built-in types have an implementation-defined range. Currently most platforms have a schema of LP64:

- `int` - 32bit
- `long long` - 64bit
- `float` - 32bit
- `double` - 64bit

There are more built-in types usable, but we do not need them right now.

Unfortunately, all built-in types usable in numeric computations convert implicitly among each other. This can lead to data loss (in case of narrowing), unexpected results or even undefined behavior. This is one of C++ weaknesses that is caused by its backward compatibility with C.

Especially using types that are `unsigned` and have a smaller conversion rank than `int` can cause grief on implicit conversions by losing their `unsigned` status and perceived wrap-around overflow behavior. Signed integer arithmetic overflow is **undefined behavior** 💣

Literals (built-in)

Literals form values of built-in types

literal	description	type
`false true`	boolean values	`bool`
`42`	decimal number	`int`
`42u`	unsigned decimal number	`unsigned`
`052`	octal number	`int`
`0x2A`	hex number	`int`
`0b10'1010`	binary number	`int`
`3.14`	floating point number	`double`
`'*'`	character	`char`
`"hello"s`	*string* literal	`std::string`

Speaker notes

Suffixes for numbers allow to modify the type of a numeric literal:

suffix	type
1L	`long`
1LL	`long long`
1u	`unsigned`
1UL	`unsigned long`
1ULL	`unsigned long long`
1.0f	`float`
1.0L	`long double`

Strings are special, because of their C legacy. Just using double quotes `"hello"` results in an array of `char` that degenerates to a pointer when used. Special suffixes are usable `"hello"s` to obtain a literal of type `std::string` or `"hello"sv` to obtain a literal of type `std::string_view`. However, those suffixes require to include the header `<string>` as well as a using directive `using namespace std::literals;` before it can be applied.

Declarations

*You introduce a name with a **Declaration***

- A declaration promises existence of **name**
- A declaration tells what **name** is:
  - *variable*: `int const a{42};`
  - *function*: `double sqrt(double d)`
  - *type*: `struct mytype;`
  - *template*: `template <typename T> ...`
- A declaration tells the type/kind of **name**

Speaker notes

Variables have a type, that is fixed, when it is declared/defined.

Functions also have a type that consists of their return type and the number and types of its parameters.

User-defined types can be class types (`struct`, `class`, `union`) or enumeration types (`enum class`, `enum`).

Templates can result in variables(constants), functions, or class types and can take constants, types, or class templates as parameters between `< >`.

Definitions

*Each **name** used must have exactly one **Definition***

***ODR**: One Definition Rule*

- Each definition is also a declaration.

Speaker notes

Definitions in header files must define only types, constants or inline functions.

The following are OK in a header file:

```
constexpr int answer{42};
constexpr int square(int x) {
  return x*x;
}
struct Count {
  int counter{0};
};
```

The following definitions are not OK in a header file:

```
int question{42}; // not constexpr
bool iseven(int x) {
  return x % 2 == 0;
} // neither constexpr nor inline
```

All translation units in a program that use a defintion of a name from a header must use the exact same definition. That is the *One-Defintion-Rule*. In addition all translation units

Variable definitions

```
type const name { value };
```

- Variable definitions occur
  - in a function body
  - as function parameter
  - as a class member variable
- **`auto const`** or **`auto`** deduces its type

Speaker notes

Variable definitions outside of functions and classes are possible but not recommended, except for compile-time constants (**`constexpr`**)

```
constexpr type name { value };
```

Instead of `type name{ inital_value };` also `type name = initial_value;` is possible. However, the latter style allows conversions with potential data loss to happen silently.

Omitting **`const`** (or **`constexpr`**) in a variable definition or declaration introduces a mutable variable. This should be rare and only be used for variables that iterate, or where the value must change over its lifetime

The use of **`auto`** for function parameter declarations is only possible from C++20 on or in lambda expressions.

Simple Function Definitions

```
constexpr
int factorial(int const n) {
  return n > 1 ?
    n * factorial(n-1) : 1;
}
```

- **`constexpr`**: function can be used at compile-time
- `return_type name(parameters) { code }`
- `condition ? when_true : when_false` if-expression (aka ternary operator)
- function definitions do not nest

Speaker notes

While recursion as shown is possible, it is often not the best solution to a problem.

A function defined with **`constexpr`** can be defined in a header file, because it is implicitly also defined as **`inline`**.

Function Defintion

```
constexpr (optional)
return-type function_name( parameters ) {
/* body */
}
```

`constexpr`
  marks function usable at compile time and implicitly **`inline`**

return-type
  every function returns a value or nothing `void`.{.cpp}
  `auto` is possible

function_name
  identifier for calling, overloading possible

parameters
  type for each parameter and name of the parameter variable zero or more parameters separated by `,`

body
  the statements of the function computing its result

Speaker notes

Function definitions often appear in implementation files, except for `constepxr` functions that are to be used by other translation units. To be called in other translation units, a function defined in an implementation file needs to be declared in a header file.

It is possible to have function definitions in implementation files that are not usable in other translation units. They don't need to have a declaration in a header file and they are usually defined within an anonymous namespace (**`namespace`** `{ /* function definition */}`).

Function definitions normally do not nest (One can use Lambdas for that).

Function Declaration

```
return-type function_name( parameters );
```

return-type
every function returns a value or nothing `void`.{.cpp}
`auto` is only possible with full function definition

function_name
identifier for calling, overloading possible

parameters
type for each parameter, optional name of parameter variable zero or more parameters separated by `,`

function signature
combination of name with parameter types, signifies overload

Speaker notes

Function declarations that are not function definitions appear in header files, for functions that are defined in an implementation file and that should be usable from other translation units. Those functions with separate definitions can not have the return type `auto` and can not be `constexpr`.

It is possible to have function definitions in implementation files that are not usable in other translation units. They don't need to have a declaration in a header file and they are usually defined within an anonymous namespace (`namespace { /* function definition */}`).

For each function signature/overload there must be exactly **one** defintion of the function in the whole program.

Lambda Function

```
[]( parameters ) {
/* body */
};
[]( parameters ) -> return-type {
/* body */
};
```

```
constexpr auto iseven { [](int i){return 0 == i % 2;} };

if (iseven(42)) { std:: cout << "even"; }
```


Speaker notes

A lambda function is an anonymous function. Technically a lambda function is an expression. To give it a name, we have to initialize a variable with it.

Not often such lambda functions are defined in a context where they are passed as arguments to other functions.

Lambda functions can have parameters of type **auto** and cannot be overloaded, because they do not have a name.

Simple Type Definition

```
struct Counter {
  int theCount{};
};
int main()
{
    Counter const one{1};
    return one.theCount;
}
```

- defines a “class-type”
- wraps an integer
- benefits:
  - no implicit conversions
  - usable as function parameter or return type

Speaker notes

https://godbolt.org/z/onjvYb8hW

`#include` Guards

- definitions in header file must not violate **ODR**

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
#include <iosfwd>
void sayhello(std::ostream &out);
#endif /* SAYHELLO_H_ */
```

Be aware that the name of the defined macro is unique per header file!

Speaker notes

Header files that have to be included in other header files can lead to violations of the **One-Definition-Rule**. To prevent such ODR violations, header files containing definitions (of templates, types, variables, or inline functions) must be protected against being `#include`d multiple times directly or indirectly.

This is solved by `#include` guards that employ conditional compilation. It is best to rely on an IDE's generation of #include guards when creating a header file.

- `#ifndef SYMBOL` - only use the following lines if the macro is **not** defined - first line in header file
- `#endif` - marks the end of the potentially excluded lines, last line in header file
- `#define SYMBOL` - defines the macro `SYMBOL` so that further #include of the same header will skip the lines with the definitions, second line in header file.

Copy-paste of code can lead to wrong `#include` guards, if the `SYMBOL` is not adjusted.

There is another commonly implemented mechanism (`#pragma once`) that is not and will not be standardized, because it will not work in more complex project setups. I recommend you stick with `#include` guards.

Function Overloads

```
constexpr
Counter factorial(Counter const n) {
  auto const count{n.theCount};
  if( count > 1 ) {
      return { count * factorial(count-1)};
  } else {
    return {1} ;
  }
}
```

- Functions with the same name can be *overloaded*, when they differ
  - in parameter type or
  - number of parameters.

https://godbolt.org/z/5Kaq6n6dW

Speaker notes

We will learn ways to write a function like factorial using your own types identically to the way one can write factorial for `int`.

https://godbolt.org/z/TKoT7zfqG

Namespaces

- namespaces group functions and types together
- namespaces nest:
  - `::` global namespace, can be omitted
  - `::std::` standard library (`std::`)
  - `std::literals` standard literal suffixes
- namespace scopes can be re-opened

namespace *name* **{** *declarations* **}**

Speaker notes

omitting the namespace name provides a special anonymous namespace. This is used to group function definitions in an implementation file that are not to be used outside of the current translation unit.

Namespaces have a special role in associating types with the functions operating on those types. Define a type and functions using the type in its parameter types in the same namespace (in general).

Expressions

Expressions denote computations of values

- each expression has a defined type
- expressions are formed from
  - operands (values) and
  - operators
- operator symbols are shorthand for functions used to compute

```
+ - * / &  ~ () [] % ^ && || ! = ?: < > <=
                    >= == !=
```


Speaker notes

most operators you might be familiar with from other languages

Peculiarities:

- Assignment (var = value) is an expression
- Parenthesis group subexpressions and denote function calls (when used after a name)
- some operators can be used *unary* and *binary*:
  - `-` x negates x
  - x `-` y subtracts
- expressions can cause ***Undefined Behavior***, e.g., division by zero

Undefined Behavior 💣

Undefined behavior

behavior, upon use of a non-portable or erroneous program construct, … for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.

- John F. Woods

Speaker notes

We will use the bomb symbol 💣 on slides when showing code that can cause undefined behavior (UB).

In general code examples that show UB or similar bad things are `marked with pinkish background`

Undefined behavior in code that can be evaluated at compile time (**`constexpr`** functions)) can be detected. However, most of the time, the actual values used in a piece of code determine if code has undefined behavior. Examples for undefined behavior are:

- division by zero (even for floating point)
- bit-shifting by more bits than the width of the type shifted
- dereferencing a **`nullptr`** pointer value

Operator overview

- arithmethic operators: `+ - * /`
- comparison operators: `== != < > <= >=`
- logical operators: `! not && and || or`
- bitwise operators: `~ | & ^ << >>`
- assignment: `= += *= /= ...`
- increment/decrement: `++ --`
- function call: `name()`

precedence mostly "normal", if unsure use parentheses

Speaker notes

C++20 introduces the three-way comparison operator `<=>` that is for defining the semantics of the other comparison operators for user-defined types. It is usually not used, unless defining such special operator overloads.

Logical and bitwise operators come in the rarely used (except for `not`) alternative spelling.

Operators have a precedence, e.g., comparison has higher precedence than logical and/or, multiplication and division have a higher precedence than addition and subtraction.

Caution when mixing types

built-in numeric types convert implicitly when mixed

In addition to surprising results this can lead to unexpected UB 💣

```
double const x { 45 / 8 }; -> x == 5.0
```

- integer division truncates
- narrowing conversion -> loss of information
- widening conversion -> can result in inaccurate values or signedness change

Speaker notes

Using `{}` for initialization (instead of `=`) prevents narrowing conversions on initialization.

Unsigned types with a lower rank than `int` (`unsigned char`, `unsigned short`) are implicitly converted to signed `int`, when used in arithmetic expression. This can cause undefined behavior on overflow, which does not occur with other `unsigned` types

```
unsigned short x{65000};
std::cout << x * x; // undefined behavior
```

Statements overview

- `;` `// empty`
- *expression* `;`
- *declaration* `;`
- `{` *statements* `}` block
- `if(`*condition*`){}else{}`
- `for(auto const`*var*`:`*range*`){}`
- `while(`*condition*`){}`
- `switch(value){ case const1: break; default:}`

https://en.cppreference.com/w/cpp/language/statements

Speaker notes

Statements are used within function bodies/blocks. Outside of function definitions, only declarations, function and type definitions are possible, but no statements. There exists the *declaration-statement* that allows declarations of local variables (and less common definition of local types). Conditional (`if`) and iteration statements group the statements in the branches or loop body in a block of statements with curly braces (`{}`). While one can elide the braces if the body or branch consists only of a single statement it is not recommended, because it easily leads to error-prone or confusing code.

This list is deliberately incomplete.

Statements are delimited by ';' unless they are a block (`{}`). The last statement in a block must be delimited `;` or block before `}`.

Variables

```
type variablename { initialvalue };
```

Best is *almost always* ***auto***:

```
auto const variable{42};
// or if initial value is fixed at compile time
constexpr auto a_constant{5};
```


Speaker notes

This is a brief overview on variables.

Note: the best variables are immutable (**const**), because they make it easier to reason about code.

Unfortunately, we have to say **const** explicitly. We only define non-const variables within function bodies or as class-type members.

Most types allow variables to be initialized to a default value by just using a pair of braces:

```
int const zero{};
```

Initialization may be omitted, but using such a variable can cause undefined behavior. 💣

Where define Variables?

- As close to its use/as late as possible!
- Consider the smallest reasonable scope { }
- A variable's lifetime ends at the end of the block it is defined in.

Speaker notes

Avoid global variables. Cevelop will warn you about

Variable naming

- Variable names can use UPPER- and lower-case letters and the underscore '_'
- The C++ convention is to start variable names with a lower case letter
- All letters in the name are significant
  - don't abbreviate `uncsrly`
- In tightly bound contexts a single letter name is acceptable

Speaker notes

Names must not start with an underscore, nor use two consecutive underscores. Those names are reserved by the C++ standard.

Mutable Variables

- non-**`const`** variables only make sense
  - for input with **`>>`**,
  - as a loop variant. and
  - when (incrementally) computing a value to be returned.
- a local variable that is returned from a function should not be **`const`**

Speaker notes

Local variables that form the return value of a function allow a compiler optimization when they are non-const. (NRVO - Named Return Value Optimization). This is suppressed when the returned variable is const.

Side effects

```
int howOldYouAre(std::istream& in, std::ostream& out){
  out << "What year were you born?";
  int year{}; // mutable
  in >> year;
  int age = 2022 - year; // mutable
  out << "Did you already have your birthday (y/n)?";
  char hadBirthday{}; // mutable
  in >> hadBirthday;
  if (hadBirthday != 'y') {
    --age;
  }
  return age;
}
```


Speaker notes

https://godbolt.org/z/hhMY3q3Mz

The code example is not splendid, because the function mixes too many things in one and also because it does not do any input failure detection.

`const` with complicated initialization

```
void sayIfItIsEven(std::ostream & out, int number){
  std::string const eventext {
    [](int number){
       if (number % 2 == 0)
         return "even";
       else
         return "odd";
     }(number) // invoke!
   };
  out << number << " is " << eventext << '\n';
}
```

IILE - immediately invoked lambda expression

Don’t use without need.

Speaker notes

Often a mutable variable with complex initialization can be defined **`const`** instead, by putting the computation of the initial value in a lambda function immediately called (IILE - immediately invoked lambda expression)

https://godbolt.org/z/q76vsojYo

However, you should not overdo it. If a value to be computed is used more than once, use a named function to compute that value instead!

Strings

```
std::string name{"Peter"};
```

- `std::string` represents a sequence of `char`
  - `"ab"` is of **NOT** type `std::string`
  - `"ab"s` is (`using namespace std::literals;`)

https://en.cppreference.com/w/cpp/string/basic_string

Speaker notes

`char` values are often only 8 bit. Only C++23 will standardize UTF-8 unicode semantics properly.

Technically string literals are arrays of char that have an extra char with the value of `'\0'`: `"Hello"` -> `char[6]`

The suffix `s` as in `"Hello"s` converts such arrays of char to the type `std::string`

Working with `std::string`

```cpp
#include <iostream>
#include <string>
void askForName(std::ostream &out){
  out << "What is your name? ";
}
std::string inputName(std::istream &in){
  std::string name{};
  in >> name;
  return name;
}
void sayGreeting(std::ostream &out,std::string name){
  out << "Hello " << name << ", how are you?\n";
}
int main() {
  askForName(std::cout);
  sayGreeting(std::cout, inputName(std::cin));
}
```


Speaker notes

https://godbolt.org/z/qYxd7fYa9

Preview References

```
void askForName(std::ostream &out)
```

- most types (including `std::string`) have *value semantics*
  - can be *passed-by value* and copied
- some objects cannot be copied
  - e.g., stream objects for I/O
- stream objects must be passed by reference (**&**)
  - this allows side effects on the original object
- references allow “Fernwirkung”

Speaker notes

for your own notes

Simple I/O

- Stream objects provide C++'s I/O mechanism
  - Pre-defined globals: `std::cin std::cout` 😟
  - Use globals ONLY in the `main()` function!
- "shift" operators read into variables or write values
  - `std::cin >> x; std::cout << x;`
- Multiple values can be streamed at once
  - `std::cout << "the value is " << x << '\n';`
- Streams state denotes if I/O was successful
  - Only `.good()` streams actually do I/O
  - You need to `.clear()` the state in case of an error

Speaker notes

for your own notes

Reading a `std::string` Value

```
std::string inputName(std::istream &in){
  std::string name{};
  in >> name;
  return name;
}
```

- successful, unless stream was `! in.good()`
- content of variable is replaced
- with no input or broken stream, string can be `empty()`

Speaker notes

for your own notes

Reading an `int` value

```
#include <istream>
int inputAge(std::istream& in){
  int age{-1}; // mutable
  if (in >> age) { // false when fail
    return age;
  }
  return -1;// input failed
}
```

https://godbolt.org/z/5onYhnhjr

Speaker notes

for your own notes

Robustly reading a number

```cpp
#include <sstream>

int inputAge(std::istream& in){
  std::string line{}; // mutable
  while (getline(in,line)){
    std::istringstream is{line};
    int age{-1}; // mutable
    if (is >> age) { // false when fail
      return age;
    }
  }
  return -1;// input failed
}
```

https://godbolt.org/z/4WxhvEsbv

Speaker notes

This example first reads a full line (or to the end of the line) and places the input in a string variable.

We then use the “trick” from our test cases to create an input stream from that string with `std::istringstream`.

On that stream we can convert the input line to a number, and if that fails, we try again with the next line.

When the input ends before we got a number, we return -1.

States of `std::istream`

```
int readFrom(std::istream & in){ . . . }
```

State bit	Query	Activated
none	`in.good()`	*initial*, `is.clear()`
`failbit`	`in.fail()`	formatted input failed
`eofbit`	`in.eof()`	trying to read at end of input
`badbit`	`in.bad()`	unrecoverable I/O error

- input on stream must check for `in.fail()` and `in.bad()`
  - if `!in.good()` you must `in.clear()` to continue input

Speaker notes

for your own notes

Dealing with invalid input

```
#include <istream>
int inputAge(std::istream& in){
  while (in.good()){
    int age{-1}; // mutable
    if (in >> age) { // false when fail
      return age;
    }
    in.clear(); // reset state
    in.ignore(); // skip 1 char
    //in.ignore(10000,'\n'); // ignore whole line
  }
  return -1;// input failed
}
```

https://godbolt.org/z/qhse5nEdz

Speaker notes

for your own notes

Formatting Output

```cpp
#include <iostream>
#include <iomanip> // setw() setprecision()
#include <cmath>   // acos()
int main() {
  std::cout << 42 << '\t'
            << std::oct << 42 << '\t'
            << std::hex << 42 << '\n';
  std::cout << 42 << '\t' // std::hex is sticky
            << std::dec << 42 << '\n';
  std::cout << std::setw(10) << 42
            << std::left << std::setw(5)<< 43 << "*\n";
  std::cout << std::setw(10) << "hallo"<<"*\n";

  double const pi{std::acos(0.5) * 3};
  std::cout << std::setprecision(4) << pi << '\n';
  std::cout << std::scientific << pi <<  '\n';
  std::cout << std::fixed << pi * 1e6 <<  '\n';
}
```

https://godbolt.org/z/qo7Ye46MG

Speaker notes

Produces the following output:

```
42  52  2a
2a  42
        4243   *
hallo     *
3.142
3.1416e+00
3141592.6536
```

Reading all characters - unformatted I/O

```cpp
#include <iostream>
#include <cctype> // for tolower()
int main() {
  char c{};
  while(std::cin.get(c)) {
    std::cout.put(std::tolower(c));
  }
}
```

https://godbolt.org/z/K7zd1x5nr

- `get()` and `put()` are unformatted I/O functions
  - What happens when we use `>>` and `<<`?

Speaker notes

- A very simple program transforming its input to lower case
  - `<cctype>` contains character conversion and character kind query functions (`std::tolower(c)`,`std::isupper(c)`)
- More in the exercises for you to experiment with!

Exercise 2

exercise02

For self study:

exercises/exercise2-extra.md

Speaker notes

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise02/

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise2-extra.md

Sequences

```
#include <vector>
#include <array>
#include <iterator>
```


Speaker notes

`std::string` works like a sequence of `char` elements.

std::vector<T>

```
std::vector v1{1,2,3,4,5};
std::vector<int> v2{1,2,3,4,5};
```

- `std::vector<T>` is a container = contains elements of type T
  - T is a template type parameter (= placeholder for type)
- `std::vector` can be initialized with a list of values
  - if empty, the element type must be specified
    - `std::vector<double> vd{};`
  - with values given, the element type is deduced
    - and the template argument angle brackets be omitted

Speaker notes

Other constructions of vector objects require the use of parenthesis instead of curly braces to enable calling the corresponding constructor. This is a idiosyncrasy that is rooted in a mistake in standardizing C++11 that cannot be undone, because of its backward compatibility goal.

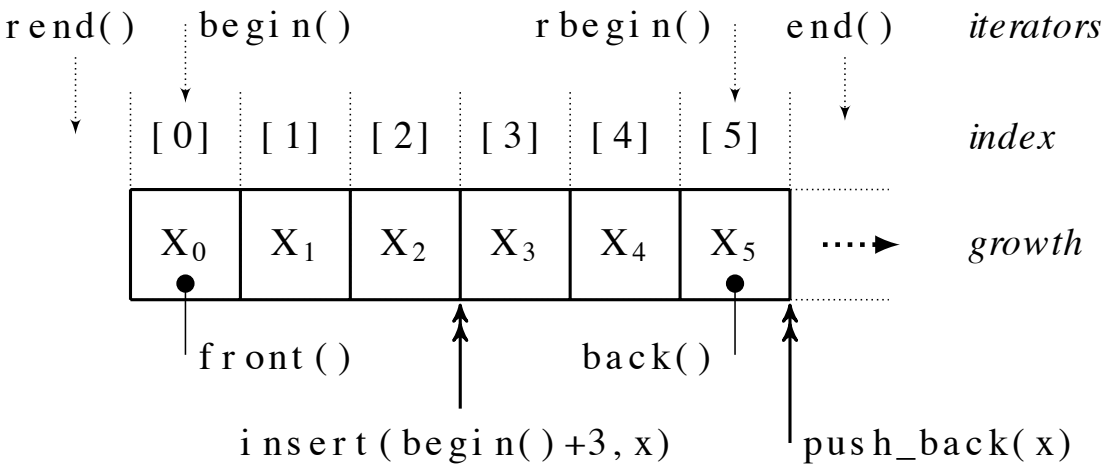
The above examples

```
std::vector{1,2,3,4,5};
std::vector<int>{1,2,3,4,5};
```

construct a `vector<int>` value with 5 elements each. To create a corresponding variable, one either can use **`auto`** or put the name in front of the initialization:

```
auto const vecint{ std::vector{1,2,3,4,5} };
std::vector<int> vecintmutable{1,2,3,4,5};
```

`std::vector<int>(6)` Overview



Speaker notes

Parentheses for intialization allow to specify the number of elements: `std::vector<int>(6)` is a vector with 6 integer elements. With braces `std::vector<int>{6}` would have a single element with the element `6`.

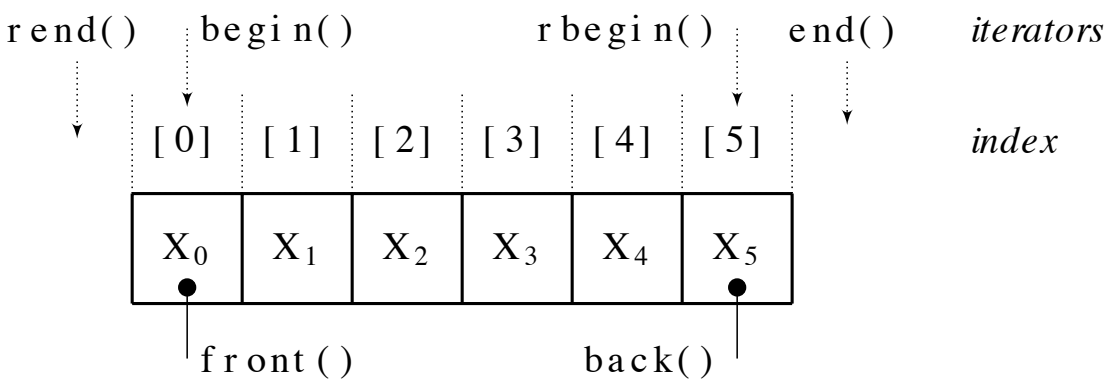
Using `push_back(value)` we can extend the vector. It will grow automatically.

The function pairs `begin(), end()` and `rbegin(), rend()` (reverse) provide iterators that allow to visit the elements in the vector sequentially, or by using index operation [n] in arbitrary order.

Accessing a vector element with indices out of bounds or an end() iterator causes undefined behavior 💣.

std::array< T, N >

fixed size sequence of elements of type T



Speaker notes

In contrast to `std::vector` the size of an array cannot change. If you have a constant vector, with just a few elements that you know the number of at compile-time, consider using `std::array` instead.

Sequence iteration (range-based **for**)

	const	mutable
copy variable	`for (auto const x : v) { std::cout << x << '\n'; }`	`for (auto x : v) { x *= 2; std::cout << x << '\n'; } // v unchanged`
reference element	`for (auto const &x : v) { std::cout << x << '\n'; }`	`for (auto &x : v) { x *= 2; } // v changed`

Speaker notes

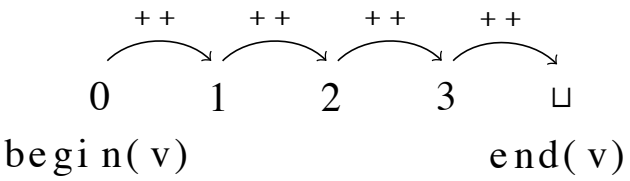
Using indices or iterators directly for iteration is not recommended.

You'll use iterators as arguments to algorithm-functions to allow access to the elements.

With "large" elements, using references for iteration can be more efficient.

C++ iterators

```
#include <iterator>
```



- iterators refer to an element in a container
- subsequent position means incrementing the iterator
- end of iteration marked by end() iterator

Speaker notes

C++ iterators do not know when the iteration has to end.

If used manually to iterate, it means that iteration could run beyond the end. 💣

A loop with manual iteration looks like the following (don't do that!)

```
for (auto it = std::begin(v); it != std::end(v); ++it) {
  std::cout << (*it)++ << ", ";
}
```

Using iterators with algorithms

```
#include <algorithm>
```

- algorithms take an iterator pair (= range) as arguments
- algorithm name tells its purpose

```
size_t
        count_blanks(std::string
         s){
  size_t count{0};
  for (size_t i{0}; i <
        s.size(); ++i){
    if (s[i] == ' '){
      ++count;
    }
  }
  return count;
}
```

```
size_t count_blanks(std::string s){
  return count(begin(s),end(s),' ');
}
```

Implementation is so short, it wouldn't even need a separate function

Speaker notes

Always prefer using algorithms over writing your own loops!

We can use functions from the namespace `std::` here (`begin(), end(), count()`) without explicitly telling the namespace. This is due to a mechanism called **Argument Dependent Lookup** or ADL. Because the argument to those functions has a type `std::string` that is defined in `namespace std`, the called functions are looked up in that namespace, resulting in `std::begin()`, `std;:end()`, and `std::count()` being called.

While often very convenient, especially finding overloaded operators, or standard library functions it can cause surprising results, when a function is found, one didn't think it exists.

`std::accumulate` and `std::distance`

```
#include <numeric>
#include <iterator>
```

summing up values in a vector (uses `+`)

```
  std::vector v{5, 4, 3, 2, 1};
  std::cout << accumulate(begin(v), end(v), 0)<< " = sum\n";
```

count elements in range (`size()` alternative)

```
void printDistanceAndLength(std::string s) {
  std::cout << "distance: "<< distance(begin(s), end(s)) <<'\n';
  std::cout << "in a string of length: "<< size(s)<<'\n';
}
```


Speaker notes

https://godbolt.org/z/sq36Mqnrf

In general, if you know the container you are operating on, you can use `size(c)` instead of `distance(b,e)`, but if you only have a pair of iterators, distance() will just work.

You can use `std::size(container)` or the container's member function `container.size()` to determine the size.

`std::for_each()`

- like range-for, uses function as third argument:

```
void print(int x) {
  std::cout << "print: "<< x << '\n';
}
void printReverse(std::vector<int> v) {
  std::for_each(crbegin(v), crend(v), print);
}
```

- 3rd argument of `std::for_each` is a function with 1 parameter of element type
- how to pass `ostream&` to print for testability?

Speaker notes

Example uses `std::crbegin()` and `std::crend()` to obtain **c**onst-**r**everse-iterators that prevent the elements iterated to be accidentally changed.

untestable, because print uses `std::cout`

https://godbolt.org/z/qh9jhanWf

Lambda with `for_each()`

```
void printReverse(std::vector<int> v, std::ostream &out) {
  std::for_each(crbegin(v), crend(v),
    [&out](auto elt){ // lambda function expression
      out << "print: " << elt << '\n';
    }); // end of for_each arguments
}
```

- Use of a variable in the lambda requires **capture**
- `[variable]` captures variable by value
- `[&variable]` captures by reference
- lambda functions allow `auto` for parameters

Speaker notes

https://godbolt.org/z/YGb557YWs

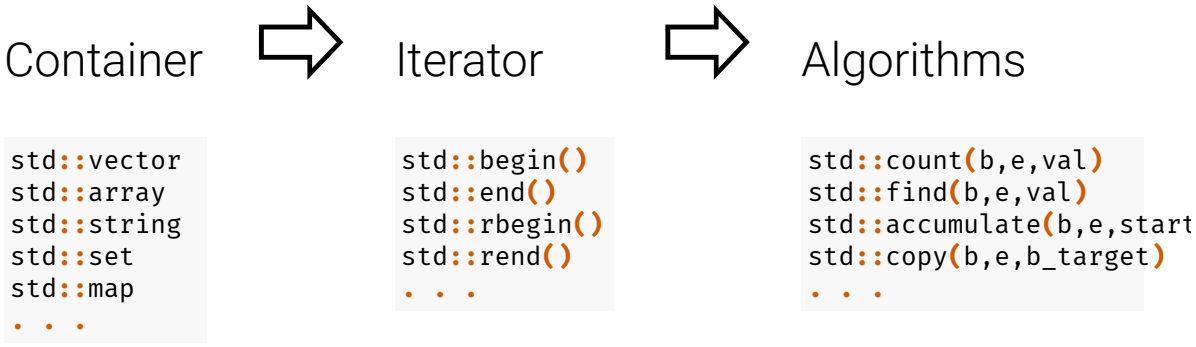
Unless a side effect on the outside variable that is captured is needed prefer capture by value.

For the stream object, we rely on a side effect, therefore, capture by reference is required.

most of the time a range-for loop is equivalent to calling the for_each algorithm, unless you want to iterate “backwards”, which is impossible with range-for (unless with a special adapter object see:

https://github.com/PeterSommerlad/ReverseAdapter

Iterators connect Containers with Algorithms



Speaker notes

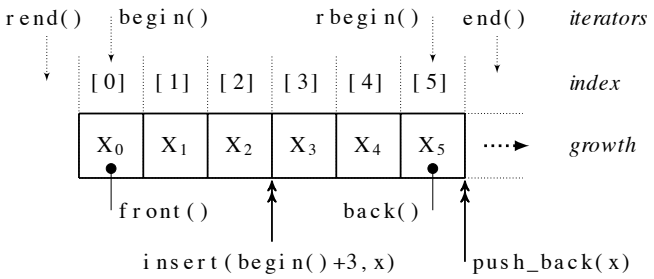
While most algorithms are defined in the header `<algorithm>` there is a significant number also defined in the header `<numeric>`

C++20 includes the “ranges” library, which is also available for C++17 as an open source library.

With ranges library, containers can be used directly with its algorithms in addition to composability of algorithms, which is not available before.

https://github.com/ericniebler/range-v3

Extending a `std::vector`



- Append: `v.push_back(value)`
- Insert: `v.insert(iterator,value);`

Speaker notes

for your own notes

`std::copy` usage

`std::copy(`input-begin`,` input-end`,` output-begin`);`

```
std::vector<int> source{1,2,3,4}, target{};
copy(begin(source),end(source),end(target)); // 💣
```

`copy()` requires output to have enough space

`std::back_inserter()` or `std::inserter()` help:

```
std::vector<int> source{1,2,3,4}, target{};
copy(begin(source),end(source),back_inserter(target)); // 💣
```

needs `#include <iterator>`

Speaker notes

https://godbolt.org/z/4z7zE4fjh

Filling a vector with same values

requires space available

```
std::vector<int> v{};
v.resize(10);
fill(begin(v),end(v),2);
std::vector<int> v(10);
fill(begin(v),end(v),2);
```

requires round parentheses for initialization

```
std::vector v(10,2);
```


Speaker notes

In addition to the fill algorithm, a vector can be initialized to have a initial value for a given amount of values. This requires round parentheses for vectors with numeric elements.

Filling a vector with generated values

`std::generate()` - needs space,
`std::generate_n()` - can use `back_inserter()`

```
generate_n(back_inserter(w),5,
  [x=1.0] () mutable {
    return x*=2.0;
  });
```

- lambdas can introduce capture variables
- changing them in the lambda body requires **`mutable`**

Speaker notes

alternative solution using a pre-sized vector and a variable captured by reference, so it can be altered within the lambda.

```
std::vector<double> powerOfTwos(5);
double x{1.0};
std::generate(powerOfTwos.begin(),
              powerOfTwos.end(),
              [&x] {return x *= 2.0;}
);
```

https://godbolt.org/z/Wdsn6hzY3

Recap: Lambdas with capture

[capture](parameters) mutable$_{opt}$ *->*
return_type$_{opt}$ *{ body }*

- *capture*: `[=]` `[&]` `[var]` `[&var]` `[var=value]`
- *paramters*: `auto p1, int p2, auto...`
- **`mutable`**: captures value variables can change in body
- *return_type*: typically omitted and deduced
- *body*: statements, return statement provides *return_type* implicitly

Speaker notes

Lambdas are implicitly **`constexpr`** and thus can be used at compile-time.

Guidelin: always capture variables explicitly by name `[=var]`, avoid capture by reference (except for streams)

Generating subsequent values

`#include <numeric>` provides access to `std::iota(b,e,startValue)`

```
int accumulateIota(){
    std::array<int,100> a{};
    std::iota(begin(a),end(a),1);
    return std::accumulate(cbegin(a),cend(a),0);
}
```


Speaker notes

Note that the function `iota()` is defined in header `<numeric>`

The code above will compute the sum of the numbers 1 to 100. Note that we use a `std::array` with a fixed size of 100 elements.

finding elements

```
    std::vector<int> v{initer{in},initer{}};
    auto found=find(cbegin(v),cend(v),42);
    if (found != cend(v)){
        std::cout << "found value " << *found << "\n";
    } else {
        std::cout << "nothing found\n";
    }
void findOdd(std::vector<int> const &v) {
    auto odd=[](int x){ return x % 2;};
    auto found=find_if(cbegin(v),cend(v),odd);
    if (found != cend(v))
        std::cout << "found odd value " << *found << "\n";
    found = find_if_not(found,cend(v),odd);
    if (found != cend(v))
        std::cout << "found even value " << *found << "\n";
}
```


Speaker notes

`std::find(b,e,value)` and `std::find_if(b,e,condition)` return an iterator to the first element that matches value or condition, if not found, they return the *end* iterator

It is important to check the returned iterator value against the end of the range given to the find algorithms to determine if an element was actually found.

The algorithm `find_if()` takes a so-called *predicate* as its third argument. This is a (lambda-)function returning **`true`** or **`false`**, where with a numeric return type (`odd` returns `1` or `0`) zero will be interpreted as **`false`** and any value that is non-zero is considered **`true`**. There are many algorithms with the suffix `_if` that follow that principle.

Guideline:
Prefer Algorithms over Loops

Speaker notes

They give better names and make it much easier to grasp.

Also, it is much easier to pass the correct arguments to an algorithms than to write the correct iteration and condition in a loop.

Speaker notes

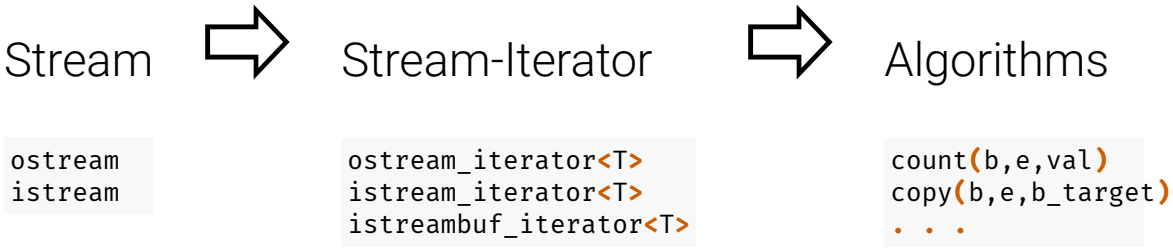
There exist more than 100 algorithm in the standard library in the headers `<algorithm>` and `<numeric>` and most of them are duplicated and some more for ranges in C++20. However, there is some structure and some of them are used only to implement other algorithms.

Nevertheless, it is good to have an overview on what is available. For example, watch the talk by Jonathan Boccara giving such an overview.

fluentcpp.com

https://youtu.be/bXkWuUe9V2I

Stream iterators for I/O




```
copy(begin(v),end(v),std::ostream_iterator<int>(std::cout,", "))
```

- `ostream_iterator<T>` outputs values of type T
- `istream_iterator<T>` reads values of type T
  - end-iterator is default-constructed: `std::istream_iterator<T>{}`
  - iteration ends when stream is no longer `good()`

Speaker notes

- `ostream_iterator<T>` outputs values of type T using `<<` to a given `ostream` with each step
- `istream_iterator<T>` reads values of type T using `>>` from a given `istream` with each step
  - end-iterator is default-constructed: `std::istream_iterator<T>{}`
  - iteration ends when stream is no longer `.good()`

`using` alias for shorter names

`std::istream_iterator<std::string>` and using it twice for input is long

`using alias_name = type;` helps to abbreviate

```
using input=std::istream_iterator<std::string>;
input eof{};
input in{std::cin};
std::ostream_iterator<std::string> out{std::cout,"\n"};
copy(in,eof,out);
```

https://godbolt.org/z/7q7YjnvKc

Speaker notes

for your own notes

unformatted input iteration

`std::istreambuf_iterator<char>` uses `std::istream::get()` for input

```
using input=std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout,"."};
copy(in,eof,out);
```

https://godbolt.org/z/o5xvnGrYP

Speaker notes

`std::istream_iterator<char>` uses >> for input, that means it will skip *whitespace* characters.

Filling a vector from input

`copy()` with `back_inserter()`:

```
std::vector<int> fillIter(std::istream &in){
    std::vector<int> v{};
    using initer = std::istream_iterator<int>;
    copy(initer{in},initer{},back_inserter(v));
    return v;
}
```

construct vector from stream input:

```
std::vector<int> fillIter1(std::istream &in){
    using initer = std::istream_iterator<int>;
    return std::vector<int>{initer{in},initer{}};
}
```


Speaker notes

for your own notes

Exercise 3

exercise03

Speaker notes

We will repeat the exercise 2 exercises, but now with algorithms instead of loops, so you can reuse your test cases.

Exercise 3.e lays the groundwork for exercise 5-b

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise03/

Functions

```
constexpr (optional)
return-type name( parameters ) {
/* body */
}
auto const name = []( parameters ) {
/* body */
};
constexpr auto name {
[]( parameters ) -> return-type {
/* body */
} };
```


Speaker notes

To be able to recall lambda functions with a name, like regular functions, we need to initialize a named variable with the lambda functions. One can use the `=` symbol or braces `{}` to initialize.

Then calling a function or a lambda function is identical, with the exception, that one cannot overload lambda functions.

The return type of a lambda function can be omitted, which has the same effect than using `auto` as a named function's return type.

Function declarations must specify the return type and cannot use `auto`.

A good Function

- does one thing well
  - and its name says it
- has only few parameters
  - and prevents misorder
- is short and simple
  - no (deep) control structures
- provides guarantees
  - and checks and reports

Speaker notes

first item corresponds to the design principle “High Cohesion”.

A good function is easy to use with all possible argument values its parameter types allow, or provides consistent error reporting if argument values prohibit delivering its result, i.e., exception.

We learn later how to prevent misordering arguments with strong types.

What do you think is a useful upper bound to the number of statements in a function?

Function Parameters

zero, one, many ?

- functions without parameters: `()`
- one parameter: `( type name )`
- two parameters: `( type1 name1, type2 name2 )`
- etc.
- parameter names can be omitted in function declarations
- like local variables, declare parameter variables as const

Speaker notes

for your own notes

Reference Parameters `(T& par)`

*Use (lvalue) reference parameters for **side effects***

- Calling the function requires a variable, not just a value
- Use non-const reference parameters only when needed
- `(T const &param)` ~~ `(T const param)`
  - const-reference parameters are an *optimization*

Speaker notes

Example: `std::string readName(std::istream &in)`

You will see a lot of code using `T const &` (const-reference to T) as the type of function parameters. This means, it will not provide the function with its own copy of the parameter, but it will also prevent alteration of the parameter value within the function. The use case for pass-by-const-reference is optimization for types T that can be expensive to copy, i.e., a `std::vector<double>` with millions of elements can be expensive to copy. Use pass-by-const-reference, when you don't need to change the parameter within a function and you don't know the parameter type or you know the parameter type can be expensive to copy. When in doubt, use pass-by-value and measure.

Function Parameter Kind

No invisible dependencies

Prefer parameter definitions as follows:

1. **pass by value**
2. *pass by const-reference* `const &` -> optimization of 1
3. pass by reference `&` -> side effect
4. (*pass by rvalue-reference* `&&` *-> transfer of ownership*)

Do not forget that you also can pass a template parameter at compile time.

Speaker notes

Some dependencies stay hidden, such as on heap availability, OS-resources, or external communication partners. Their failure mode is discussed below.

Prefer pass by value, unless the type is really expensive to copy, i.e., a large std::vector or std::array. Then, pass by const-reference.

When you need a side effect on specific object or the object cannot easily be copied or moved, and only then, pass by non-const reference.

Always consider implementing pure function, returning their result based on a parameter instead of a side effect on the parameter.

Passing by r-value reference is for taking ownership. This is a topic for C++ Advanced.

Passing by forwarding reference (deduced r-value reference syntax) is for perfect forwarding. This is a topic for C++ Expert.

Parameter Styles Overview

	non-const • **mutable parameter**	**const** • **parameter can't change**

copy:
• parameter is local variable
	`void f(std::string s) {`
	`  //modification possible`
	`  //side-effect only locally`
	`}`
		`void f(std::string const s) {`
		`  //no modification`
		`  //used for maximum constness`
		`}`

reference:
• call-site argument access
	`void f(std::string & s) {`
	`  //modification possible`
	`  //side-effect also at call-site`
	`}`
		`void f(std::string const & s) {`
		`  //no modification`
		`  //optimization for large objects`
		`}`

Speaker notes

Using **`const`** on a by-value parameter has no effect on the call site. The **`const`** only means, that the local variable for the parameter is immutable. This **`const`** is often omitted.

The **`const &`** on a reference parameter is significant, because it allows the same call-site as a pass-by-value parameter.

A non-const reference parameter requires a variable as an argument at the call site.

Returning from a Function

`return` *expression*`;`
`void` return-type: **`return`**`;` or `}` at end of function

- **`return`** defines value to be returned and exits function
- multiple **`return`** statements are possible
  - return-type **`auto`** requires all **`return`** statements to return an expression of the same type
- executing a return statement leaves the function and execution continues at the call site
- passing `}` at the end of a non-`void` function is UB 💣

Speaker notes

Don't be afraid to return a large object that is expensive to copy from a function, a compiler will normally optimize that situation.

A function with a `void` return type only makes sense, when it has a side effect. So it must need a reference parameter.

Function Styles

pure	**vs**	**side-effect**
always same result with same argument		state dependent results vary over time

- Pure functions test easily
- Functions with side-effect should have target as parameter, not a global.
- Lambda functions can have side-effects:
  - with reference capture
  - with capture and **`mutable`**

Quiz: Which side effects on global state do you know?

Speaker notes

Pure functions can easily tested in isolation.

Side-effect functions that have side-effects on non-arguments are hard to test.

Pass all side-effect targets as parameter, when possible.

Parameter needs to support abstraction for better testability, e.g., using `std::ostringstream` vs `std::cout` for tests.

Hidden side effects can be: allocating/deallocating memory, spawning a (detached) thread, calling the OS/i.e. I/O.

Testing a function

- pure function needs tests for input arguments
  - categorize arguments
    - 0, 1, many
    - success or failure
  - exhaustive tests are impossible
- impure functions need tests also for different initial dependent state
  - much harder to test
  - impossible with global variable dependency

Speaker notes

With a pure function a test is easier to formulate, because a call with the same argument values will deliver the same result.

Impure functions rely not only on parameters but have additional state that influences their result or behavior. This makes testing much harder.

To write good test cases for a function, it is important to understand, why it can fail. Not all functions can succed with all possible argument values.

Function Overloading Ambiguity

beware of implicit conversions

```
2  int factorial(int n){
3    if (n > 1)
4      return n *
     factorial(n-1);
5    return 1;
6  }
7  double factorial(double
     n) {
8    if (n < 15)
9      return
     factorial(int(n));
10   double result=1;
11   while(n > 1) {
12     result *= n--; //
     bad!
13   }
14   return result;
```

```
16 int main() {
17   //factorial(10u); //
     ambiguous
18   //factorial(1e1l);//
     ambiguous
19   std::cout << factorial(3) <<
     "\n";
20   std::cout << factorial(1e2)
     << "\n";
21 }
```


Speaker notes

https://godbolt.org/z/MhhK6Wo7d

observe the ambiguity by commenting out the lines 17 and 18.

What is “bad” on line 12?

Function default Arguments

```
void increment( int & var, unsigned delta = 1) {
  var += delta;
}
```

- rightmost parameters can have default values given with **=**
- implicit overload with fewer parameters
- declaration can define default argument
- definition can omit default argument

```
int couter{0};
increment(counter); // uses default argument with value 1
increment(counter,5);
```


Speaker notes

Use default arguments with thought. Sometimes it is better to provide two overloads where the one with fewer parameters passes the default arguments to the body of the function with more parameters:

```
void increment( int & var, unsigned delta) {
  var += delta;
}
void increment( int & var ) {
  increment(var, 1);
}
```

Functions as Parameters: g(T f(T))

Functions are first class objects

```cpp
#include <iostream>
#include <cmath>
void applyAndPrint(double x, double f(double)) {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}
int main() {
  auto const times_three = [](double value) {
    return 3.0 * value;
  };
  applyAndPrint(1.5, times_three);
  applyAndPrint(0.0, acos); // π/2
}
```


Speaker notes

Drawback: A function parameter declared like this, does not accept a lambda with a capture, since this is more than just a function.

https://godbolt.org/z/hq4E1EW6W

The following would not compile:

```cpp
int main() {
  double factor{3.0};
  auto const multiply = [factor](double value) {
    return factor * value;
  };
  applyAndPrint(1.5, multiply);
}
```

Technically such a function type parameter is a function pointer (or function reference). Calling a function parameter looks like a regular function call.

std::function<T(T)>

std::function allows more flexible function parameters

```
void applyAndPrint(double x, std::function<double(double)> f) {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}

int main() {
  double factor{3.0};
  auto const multiply = [factor](double value) {
    return factor * value;
  };
  applyAndPrint(1.5, multiply);
}
```


Speaker notes

`std::fucntion` objects can actually be re-assigned or even be “empty”. This needs to be taken care about.

https://godbolt.org/z/G3ebfv6ne

Lambda auto-Parameter for functions

A Lambda's auto parameter can be used to pass any kind of function:

```
auto const applyAndPrint{
    [](double x, auto f) {
       std::cout << "f(" << x << ") = " << f(x) << '\n';
    }};
int main() {
  double const three{3.0};
  auto const times_three = [three](double value) {
    return three * value;
  };
  applyAndPrint(1.5, times_three);
}
```


Speaker notes

A lambda's auto parameter is a "generic" parameter. Unfortunately, only C++20 allows such auto Parameters for regular functions.

https://godbolt.org/z/K7bocs9bW

As of C++20 a normal function can use auto parameters as well:

https://godbolt.org/z/Pa6aWrcM6

Technically functions with auto parameters are function templates.

Function Contracts

Preconditions

What a caller needs to ensure

What a function can check

Postconditions

What the functions guarantees

Fault

Pre-Condition violated by caller

Postcondition cannot be satisfied

Error

Fault is detected

Speaker notes

A precondition could be violated, because the caller did not provide a value in a correct range:

- negative or too big index
- divisor is zero

A postcondition could not be satisfied, because resources for the computation cannot be aquired:

- Out of memory
- File to open not found

Dealing with Errors

Your function's contract is violated

0. **ignore faults** and eventually have *undefined behavior*
1. return a **standard result** to cover the error
2. return a special **error value**
3. provide an ***error status as a side-effect***
4. throw an **exception**

But first you must detect the fault.

Speaker notes

Be aware of your function's contract, even if you don't state it explicitly

-1. Always Succeed

without preconditions and without possibility of violating its own postcondition a function is the simplest to use.

- unfortunately, not all functions we write can work with all argument values

Speaker notes

A (pure) function that has a well-defined result for all its possible argument values can always succeed.

Technically, one can mark such functions with the keyword **`noexcept`** (see C++ Advanced.)

0. Ignore Faults

```
std::vector v{1, 2, 3, 4, 5};
v[5] = 7;
```

- Caller is a faultless super-hero 🦸
- Most efficient, no checks
- Simple to implement, but hard to use

Do it consciously and consistently

Better also provide a safe alternative

`std::vector::at()`

Speaker notes

Ignoring possible faults will lay the burden on the function caller. A lot of C++'s legacy from C relies on the programmer being a super hero 🦸 and always call a function or use an operator correctly. Doing otherwise, leads to undefined behavior 💣!

For small exercises, it can be OK to ignore faults, but in general it is unwise if there are no sanity checks. Especially, when function arguments rely on user input. Humans will make errors (but hardware as well :-). Assumptions about the caller and the arguments passed, are better enforced, either at compile time, by using appropriate types (that might involve checking), or at run-time with additional precondition checks.

1. Cover Error with Standard Result

```
std::string inputName(std::istream & in) {
  std::string name{};
  in >> name;
  return name.size() ? name : "anonymous";
}
```

- Caller relieved from checking
- Can hide underlying problems (harder to debug)
- Better when caller can provide own error value:

```
std::string inputName(std::istream & in, std::string def="anonymous") {
  std::string name{};
  in >> name;
  return name.size() ? name : def;
}
```


Speaker notes

While it is still easy to use as ignoring faults, it can hide problems, because things always seem to work.

2. Special Error Value

- Feasible, when return type has unused “error” values
- Sometimes invented: `std::string::npos`

```
bool contains(std::string const & s, int number) {
  auto substring = std::to_string(number);
  return s.find(substring) != std::string::npos; // HERE
}
```

- POSIX system calls use `-1`, `nullptr`, or?
  - `mmap`: `reinterpret_cast<void*>(-1)`
- **Caller needs to check!**
  - Danger of ignoring significant errors

Speaker notes

For example, who checks the return value of `::write` or `printf`?

The `end()` iterator returned from the `find()` algorithm is another example for marking an error.

2.a Modern C++ Special Error Return

```
std::optional<std::string> inputName(std::istream & in) {
  std::string name{};
  if (in >> name) return name;
  return {};
}
```

- `std::optional<T>` extends `T` with an “empty” value
- If caller doesn’t check -> UB or exception
  - check validity before `*opt` or use `.value()`

```
int main() {
  std::optional name = inputName(std::cin);
  if (name) { std::cout << "Name: " << *name << '\n'; }
}
```


Speaker notes

Note: `boost::optional<T>` can be a better choice, because `boost::optional` supports optional references. The standard library version unfortunately not (yet?).

`std::optional` is also great for functions with optional parameters, that either might be present or not. The called function otoh, must decide what to do with a missing parameter value, the empty `optional` argument.

2.b Modern C++ Special Error Return

- `std::variant<T,EC>`: result or error code
- transports more information than just *oops*

```
enum class error_code { empty, invalid};
std::variant<std::string, error_code> inputName(std::istream &is){
  std::string input{};
  is >> input;
  if (input.size()){
    if (all_of(cbegin(input), cend(input), isalpha)) {
        return input;
    } else { return error_code::invalid; }
  } else { return error_code::empty; }
}
```

- a dedicated error-code keeping type `std::expected` will appear in C++23

Speaker notes

`std::expected<T,E>` was just accepted for the C++ standard library in C++23 (2022-02)

Using `std::variant<T,E>`

```
int main(){
  auto result = inputName(std::cin);
  while(std::holds_alternative<error_code>(result)){
    switch(std::get<error_code>(result)){
    case error_code::empty: std::cout << "missing input\n"; return 1;
    case error_code::invalid: std::cout << "wrong characters\n"; break;
    }
    std::cout << "\nplease try again: \n";
    result = inputName(std::cin);
  }
  if (std::holds_alternative<std::string>(result)){
    std::cout << "you are " << std::get<std::string>(result);
  }
}
```


Speaker notes

Using std::variant can be a bit involved, if not using `std::visit` and `overloaded`:

https://gcc.godbolt.org/z/nEeEKMEzd

3. Error Status as Side-effect

- Requires reference parameter
  - e.g. **`this`** of member functions
  - annoying when error variable is required
  - example: **`std::istream`** objects
- **BAD:** global variable 🤮
  - POSIX' **`errno`** is the *glorious* example

```
std::optional<int> inputNumber(std::istream &in){
  int number{};
  in >> number; // error as side effect
  if (in.fail()) {
    in.clear(); // reset error state
    return std::nullopt;
  }
  return number;
}
```


Speaker notes

for your own notes

4. Exceptions

- only means for constructors (and operators) to fail
  - when class invariants cannot be established
- handle, propagate or terminate (noexcept)
- allow error handling further up the call chain

Throw by value, catch by const-reference

- standard exception hierarchy is not always useful
- exceptions should not be used for expected errors
  - I/O errors, user input, environment
- error-handling (`catch`) is separated

Speaker notes

Exceptions can be great or a great burden.

Their design and implementations are old, we no would know better, but changing it is intrusive.

- relatively expensive at run-time
  - mostly due to exception-unfriendly ABIs
- except for those named `bad_*` standard exceptions not perfect
  - some allocate a message string (`logic_error, runtime_error`)
  - which can fail as well…
  - cannot be “consumed” by move
- throwing across threads possible
  - but be aware of data races (catch by const-reference)

`throw` expression

```
throw 15; // any copyable type -> int
throw std::invalid_argument("wrong");
```

- Exception thrown will propagate up the call chain
  - until suitable `catch()`
  - or terminate the program
- no specification what could be thrown
- no stack trace or source position available
  - but could be implemented (C++23)

Speaker notes

C++23 will have `std::stacktrace` to collect information on the call chain usable with an exception.

Catching Exceptions

- **try-catch** block
  - **catch** match by type
- **throw** by value, **catch** by const-reference
  - allows polymorphic exception types
  - optimizes away copy
- first **catch()** wins
- **catch(...)** -all last
- rethrowing exception up with **throw;**

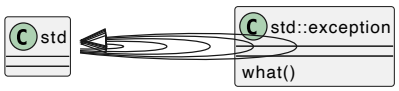
```
try {
  throwingCall();
} catch (type const & e) {
  //Handle type exception
} catch (type2 const & e) {
  //Handle type2 exception
} catch (...) {
  //Handle other exception
        types
  throw; // up the call
        hierarchy
}
```


Speaker notes

Much more on exceptions and exception safety in: C++ Advanced and C++ Expert

`std::exception` Type Hierarchy

```
#include <stdexcept>
```



- subclasses have `reason` parameter
- `what()` member function to obtain reason

Speaker notes

for your own notes

standard exception **`catch`** example

```
std::string inputName(std::istream &is){
  std::string input{};
  is >> input;
  if (input.size()){
    if (all_of(cbegin(input), cend(input), isalpha)) {
        return input;
    } else { throw std::runtime_error("wrong character"); }
  } else { throw std::logic_error("missing input"); }
}
int main(){
  while(std::cin) {// eof terminates
    try {
      auto result = inputName(std::cin);
      std::cout << "you are " << result << '\n';
    } catch (std::logic_error const &e){
      std::cout << e.what() << '\n';
    } catch (std::runtime_error const &e){
      std::cout << e.what() << '\n';
    }
  }
}
```


Speaker notes

https://godbolt.org/z/MWh5YGzd3

Testing Exceptions: ASSERT_THROWS

```
double square_root(double x) {
  if (x < 0)
    throw std::invalid_argument{"square_root imaginary"};
  return std::sqrt(x);
}
void testSquareRootNegativeThrows(){
  ASSERT_THROWS(square_root(-1.0),std::invalid_argument);
}
```


Speaker notes

`ASSERT_THROWS(expr,exception_type)` checks if the given expression will throw and exception of the given type. It will be unsuccessful if the expression doesn't throw and exception or if it throws a value of a different/incompatible type.

You cannot test for a specific exception object content with `ASSERT_THROWS`

Testing Exceptions manually

```
void testForExceptionTryCatch(){
  std::vector<int> empty_vector{};
  try {
    empty_vector.at(1);
    FAILM("expected Exception");
  }
  catch (std::out_of_range const&){} // expected
}
```

This approach allows for checking the exception thrown for details.

Speaker notes

When no exception is thrown, `FAILM(message)` will make the test case fail.

When an exception of the caught type is thrown, the test case exits without raising an error.

Testing Exception Object

```
void testForExceptionTryCatchMsg(){
  std::vector<int> empty_vector{};
  try {
    empty_vector.at(1);
    FAILM("expected Exception");
  }
  catch (std::out_of_range const&e){
    ASSERT_EQUAL("",e.what());
  } // expected
```

This approach allows for checking the exception thrown for details.

Speaker notes

When no exception is thrown, `FAILM(message)` will make the test case fail.

When an exception of the caught type is thrown, the test case exits without raising an error.

Exercise 4

exercise04

Speaker notes

Exercise 4 uses what we just learned on passing functions as parameters

When you finish early, catch up with the previous exercises, especially the word list.

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise04/

User-defined Class Types

- `struct`
- `class`
- (*`union`*)

Speaker notes

We won't treat the keyword `union`. Use `std::variant` for so-called "sum-types". If you want to have a single variable that could hold a value from one of a fixed set of different types.

A good Class

- does one thing well
  - and its name says it
- consists of member functions with only a few lines
  - avoid deeply nested control structures
- has a class invariant and guarantees it if needed
  - provides guarantees about its state
  - constructors establish that invariant
- Is easy to use without complicated sequencing requirements

Speaker notes

for your own notes

Simple Class-type Definition

```
struct Counter {
  int theCount{};
};
int main()
{
    Counter const one{1};
    return one.theCount;
}
```

- defines a “class-type”
- member-variable: `theCount`
- `;` semicolon to mark the end!
- benefits over using int:
  - no implicit conversions
  - usable as function parameter or return type

Speaker notes

https://godbolt.org/z/onjvYb8hW

I prefer the keyword `struct` over the almost equivalent keyword class, because that shows the public parts of a type with less code. Usually the public parts should come first, they are the most interesting for a user of the type.

We will incrementally extend our type `Counter`.

Do not forget the semicolon at the end of a class definition. It is required to mark the end of the definition/declaration and separate it from the next. Try in the exercises what happens if you forget that semicolon.

Class with Associated Functions

```
namespace first {
struct Counter{
  int theCount{};
};
inline
void increment(Counter &c){
  ++c.theCount;
}
inline
void print(std::ostream &out, Counter const c) {
  out << c.theCount;
}
}
```

put class and associated functions in the same namespace for ADL

Speaker notes

ADL - Argument-Dependent Lookup for finding function definitions depending on their argument type.

This example defines the `print` and `increment` functions as inline functions to enable their implementation in a header file.

Testing `first::Counter`

```
void CounterConstruction() {
    first::Counter c{42};
    ASSERT_EQUAL(42,c.theCount);

}
void
        CounterDefaultConstructi
        {

    first::Counter c{};
    ASSERT_EQUAL(0,
        c.theCount);
}
```

```
void
        CounterIncrementIncrementsByOn
         {
    first::Counter c{42};
    increment(c);
    ASSERT_EQUAL(43, c.theCount);
}
void CounterCanPrintItsValue(){
    std::ostringstream out{};
    first::Counter const c { 5 };
    print(out,c);
    ASSERT_EQUAL("5",out.str());
}
```


Speaker notes

for your own notes

Class type with Member Functions

```
namespace second {
struct Counter {
  int theCount { };
  void increment() & {
    ++theCount;
  }
  void print(std::ostream &out) const {
    out << theCount;
  }
};
}
```

*member functions implicitly take an object parameter named ***this** and have direct access to other members.*

Speaker notes

An alternative to associate functions with a class type is to create the functions as **member functions**.

Calling a member function uses an object of the type, which is the implicit first argument to the function and a dot between the object and the function name:

```
Counter c{42};
c.increment();
c.print(std::cout); // prints 43
```

Technically, those member functions are called *non-static member functions*. We will see *static member functions* in C++ Advanced.

Instead of using a member name directly, one can also write `this->membername` This notation is needed in generic (template) code to resolve name lookup ambiguities, which would select the wrong entity in the body of the class template member function.

The “name” `*this` comes from a very early time in the creation of C++, where C++ did not have references, but the concept and syntax for pointers inherited from C. Pointers have a de-referencing operator (unary `*`) that provides access to the referred (pointed-to) object. So syntactically `this` is a pointer, even if it should be considered a reference named `*this`. Pointers will be introduced in C++ Advanced.

Marking side-effects on `*this`

```
void increment(Counter &c){
  ++c.theCount;
}
```

The reference parameter indicator **&** for **`*this`** moves after the member function parameter list:

```
void increment() & {
  ++theCount;
}
```


Speaker notes

This *ref-qualification* of a member function is a feature introduced with C++11.

Most code still does not explicitly mark the member functions with a side effect on **`*this`**.

However, this introduces a hole in the C++ type system and an inconsistency with regular parameters. Therefore, use & to mark member functions that have a side effect on the class' object.

Marking `const`-ness on `*this`

```
inline
void print(std::ostream &out, Counter const c) {
  out << c.theCount;
}
```

The `const` for `*this` moves after the member function parameter list:

```
void print(std::ostream &out) const {
  out << theCount;
}
```


Speaker notes

Member functions that do not change the `*this` object, are marked with `const` after the member function parameter list. This corresponds to passing the `*this` object as an implicit const-reference parameter to the member function.

In case you provide overloads of a member function in a class for both cases, mutable and const *this, then the const-member function must be qualified with `const &` as well, or, older style, you omit the ref-qualification at the member function overload that does not have `const`. The restriction that either all overloads of a member function must be ref-qualified or none of the overloads will be lifted again in C++23 (probably).

Testing `second::Counter`

```
void CounterConstruction() {
    second::Counter c{42};
    ASSERT_EQUAL(42,c.theCount);

}
void
        CounterDefaultConstructi
        {

    second::Counter c{};
    ASSERT_EQUAL(0,
        c.theCount);
}
```

```
void
        CounterIncrementIncrementsByOn
         {
    second::Counter c{42};
    c.increment();
    ASSERT_EQUAL(43, c.theCount);
}
void CounterCanPrintItsValue(){
    std::ostringstream out{};
    second::Counter const c { 5 };
    c.print(out);
    ASSERT_EQUAL("5",out.str());
}
```


Speaker notes

Note, that instead of passing the Counter object as a function argument (in parenthesis), we name the counter object `c`, use a dot `.` and then the member function to call it: `c.increment(); c.print(out);`

Separate Member::Function Definition

```cpp
#include <iosfwd>
namespace second_separate_impl {
struct Counter {
  int theCount { };
  void increment() &;
  void print(std::ostream &out) const;
};
}
```

```cpp
#include "Counter.h"
#include <ostream>
namespace second_separate_impl {
void Counter::increment() & {
  ++theCount;
}
void Counter::print(std::ostream &out)
        const {
  out << theCount;
}
}
```

separate member function definitions need to prefix the name with the class name: ***`Counter::increment`***

Speaker notes

To lessen the dependency on other header files, we can actually split the member function definition from the class definition.

Encapsulating class Members

- **`struct`** provides **`public:`** visibility
- **`class`** defaults to **`private:`** visibility
  - only other class members and **`friend`** functions can access the member
- **`public:`** **`private:`** mark the visibility of the following class members

```
private:
  int theCount { };
};
```

*private data members require initialization, either directly or through a **constructor***

Speaker notes

I personally prefer using **`struct`** to introduce a class type, because the public members are more important and should come first.

Others might prefer **`class`** to denote, that the type has member functions (in contrast to C's **`struct`**) or that the type has encapsulated **`private:`** members.

Defining a Constructor

```
explicit constexpr
Counter(int const initial =
          0)
: theCount{initial} { }
```

```
private:
  int theCount { };
};
```

```
class IntPair {
  int first{};
  int second{};
public:
  IntPair(int f, int s)
  : first{f}
  , second{s}
  { }
};
```

- same name as the class type
- looks like a function without return type
- data members with `:` after parameters
- empty function body `{}`
- **`explicit`** for *ctors* with single parameter
- **`constexpr`** for use at compile-time

Speaker notes

With multiple non-static data members, the initialization happens in the sequence of their definition within the class type. Therefore, define the data members and list their initialization after the `:` (colon) in the same sequence.

Constructor is often abbreviated with *ctor*.

You can define multiple constructor overloads.

Here the constructor with a default argument actually defines two overloads:

- one callable without arguments (= default constructor)
- one callable with a single `int` argument (= conversion constructor)

Any constructor callable with a single argument must be marked **`explicit`**, because otherwise, it might be applied silently by the compiler for conversions, leading to surprises.

In addition to initialization, a constructor can establish a `class invariant`, i.e., a combination of data member values that fulfils a required condition. All mutating member functions will then guarantee that this condition is never violated (=invariant). For example, one could guarantee that the counter is never negative. Constructors establishing a class invariant usually have a non-empty function body and throw an exception, when the class invariant would be violated.

Default Constructor =default

Defining any constructor removes ability initialize with {}:

```
  explicit constexpr
  Counter(int const initial)
  : theCount{initial}{}
```

```
void CounterDefaultConstruction(){
  Counter c{};
  ASSERT_EQUAL(Counter{0}, c);
}
```

fails to compile, unless default constructor is resurrected:

```
  constexpr
  Counter() = default;
```


Speaker notes

In our previous example

The `= default;` resurrects the constructor without parameters that the compiler would normally provide for a class. It is safe to do so, because we initialize the sole data member variable with zero:

```
protected:
  int theCount { };
};
```

How to Test encapsulated Members ?

We can no longer access the value of the private data member in the class type for our tests.

```
ASSERT_EQUAL(43, c.theCount);
```

fails to compile, because the member `.theCount` is **`private:`**.

```
ASSERT_EQUAL(Counter{42},c);
```

fails, because we cannot (yet) use **`operator==`** with `Counter` objects.

Speaker notes

There are several options:

1. define the Test functions as `friend` of the class `Counter`. This couples the class type to all its test functions
2. Only test via the externally visible behavior, i.e., print(). This is actually in general the preferred way!
3. Provide the equality comparison operator for our class Counter.

The latter option either requires a member function or a friend function that can access the data member.

Defining equality comparison (pre C++20)

```
constexpr friend bool
operator==(Counter const &left, Counter const &right) {
  return left.theCount == right.theCount;
}
```

- implement a function called `operator==` with two parameters
  - arguments passed by **`const&`** (or by value)
  - within the class definition as **`friend`**
- `bool` return type
- to allow use at compile time: **`constexpr`**
- to access the private data member: **`friend`**

Speaker notes

While it is not strictly necessary, passing the arguments to the comparison operator by const-reference is a typical pattern. Follow it, because it will work also well in cases, where the class is “big”.

A friend function must be declared within a class type and it is allowed access to **`private:`** members of the class. Putting the actual definition of the friend function within the class type, makes it special as a:

Hidden Friend

The concept of implementing `friend` functions within a class definition leads to a beneficial situation with overload resolution and can make code compile faster: The overload will only be considered by the compiler, when one of the arguments has the actual type of the parameter. Otherwise, potential implicit conversions may lead to multiple overload candidates to be resolved for each comparison.

The **`constexpr`** marking on our operator function is optional in this case (see C++ Advanced for compile-time programming) but I recommend that you prepare your own types that do not need run-time features to be useful for compile-time computation.

Consistent equality comparison (pre C++20)

```
constexpr friend bool
operator !=(Counter const &left, Counter const &right) {
  return !(left == right);
}
```

- implement a function called **`operator!=`** with two parameters
  - like **`operator==`**
- delegate to **`operator==`** and logically negate (**`not`** or **`!`**) its result

Speaker notes

The boost.operators library provides features for consistent definition of related operators (i.e., **`==`** and **`!=`**, or **`< > <= >=`**). Using that library is a topic for C++ Advanced.

The alternative spelling for logical negation **`not`** can be used to increase readability to the operator symbol **`!`**. **`not=`** is not allowed, unequal musst be spelled **`!=`**

Testing with Encapsulation

```
using third::Counter; // Counter
        visible
void CounterConstruction() {
  Counter c{42};
  ASSERT_EQUAL(Counter{42},c);
  // uses operator==
}
void CounterDefaultConstruction(){
  Counter c{};
  ASSERT_EQUAL(Counter{0}, c);
  // uses operator==
}
```

```
void CounterIncrementIncrementsByOne() {
  Counter c{42};
  c.increment();
  ASSERT_EQUAL(Counter{43}, c);
}
void CounterCanPrintItsValue(){
  std::ostringstream out{};
  Counter const c { 5 };
  c.print(out);
  ASSERT_EQUAL("5",out.str());
}
```


Speaker notes

The `using third::Counter;` is called a *using-declaration*. It imports the name `Counter` from the namespace `third` into the current namespace. The test cases for each class variant have their own namespace each. this allows to reuse the function names.

Equality Comparison C++20

C++20 feature: defaulted comparison operators.

```
constexpr
bool operator==(Counter const &other) const = default;
```

- operator==() member function -> 1 Parameter + implicit *this parameter
- must return `bool`
- must take implicit `*this` as `const`
- must take class type as single explicit parameter
- member `operator==` provides `==` and `!=`

Speaker notes

If you can use C++20 already, it is beneficial to rely on defaulted comparison operators for your type, because it is less code and less error prone to obtain comparison operators.

While techically possible to define defaulted equality comparisons between different types (i.e., `Counter` and `int`), I do not recommend this.

`operator` Overloading

We can overload most of the built-in operators to be useful for user-defined types

binary @ : `+ - * / % == != < > <= >= | & ^`

```
return-type operator@(T,T)
return-type operator@(T) as member function
```

unary operatators @ : `+ - ! ~ ++ -- *`)

```
return-type operator@(T)
return-type operator@() as member function
```


Speaker notes

Whenever you decide to overload operators for your type, ensure that implicit conversions do not lead to ambiguities and stick to homogeneous overloads (don't allow adding apples and oranges, for example).

I also recommend that you overload unary operators as member functions and binary operators as *hidden friends*.

Most common usage of operator overloads are for comparison and input/output: `operator<< operator>>`

The above list of overloadable operator symbols is incomplete.

You cannot invent own operator symbols or change the underlying grammar/operator precedence rules in C++.

Increment `operator++`

How to distinguish `++`prefix and postfix`++`?

```
Counter& operator++() & { // ++c
  ++theCount;
  return *this;
}
```

- as a unary operator prefix **`operator++`** has only one parameter.
- postfix`++` has an additional unnamed parameter of type `int`:

```
Counter operator++(int) & { // c++
  Counter result{*this}; // save
  ++ *this; // delegate to prefix
  return result;
}
```


Speaker notes

Increment and decrement operators take their argument by reference, to enable the side effect of changing the variable.

Prefix **`operator++`** and **`operator--`** can return a reference to the changed parameter, because they will always be applied to variables and never plain values or temporaries. Note: return by reference is in general a bad idea, because it can easily lead to dangling and thus undefined behavior 💣.

Postfix **`operator++`** and **`operator--`** return the previous old value. It is common practice to implement them based on the user-defined prefix operator and by storing the original value of the parameter and return that original value.

The additional parameter of type int that marks the postfix version will never be used, so it does not need a name. It was just “invented” to disambiguate that special case of `++` and `--`.

replacing `print()` with `<<`

```
friend
std::ostream& operator<<(std::ostream &out, Counter const&c) {
  return out << c.theCount;
}
```

Output stream operators take the stream object by reference as their first (left) parameter. Therefore, they must either be members of `std::ostream` or non-member functions.

To allow operator chaining, overloaded stream operators return their stream reference parameter by reference.

Speaker notes

Reading from a stream similarly would overload `std::istream& operator>>(std::istream& in, MyClass &object)`.

You need to pass both parameters of an input operator by reference, to achieve the side effects:

- std::istream& - input on the stream side effect
- MyClass& - changing the value on successful read

Ordering `Counter`

How to compare Counters?

```
void CounterCanCompareLess() {
  ASSERT(Counter{} < Counter{1});
}
void CounterCanCompareLessEqual() {
  ASSERT(Counter{} <= Counter{1});
}
void CounterCanCompareGreater() {
  ASSERT(Counter{2} > Counter{1});
}
void CounterCanCompareGreaterEqual() {
  ASSERT(Counter{2} >= Counter{1});
}
```


Speaker notes

It would be great, if we could also decide wether one Counter is greater or less another Counter.

Ordering easy with C++20

- C++20 introduces a new **operator<=>**
- *Three-way-comparison* operator

```
constexpr
auto operator<=>(Counter const &other) const = default;
```

- provides **< > <= >=** comparison operators
- return type **auto**
- parameter tyes as with **operator==**
- **= default**

Speaker notes

The details of the three-way comparison will be covered in C++ Expert/Advanced.

The three-way comparison can be implemented as a member function, it will still be applied in all situations, even when conversion (which is not allowed) is happening on the left-hand operand.

Ordering consistent in C++17

- do: **`operator<(T const &l, T const& r)`**

```
constexpr friend bool
operator<(Counter const &left, Counter const &right) {
  return left.theCount < right.theCount;
}
```

- and delegate all others to it (**`> >= <=`**)

```
constexpr friend bool
operator>=(Counter const &left, Counter const &right) {
  return !(left < right);
}
constexpr friend bool
operator>(Counter const &left, Counter const &right) {
  return (right < left);
}
constexpr friend bool
operator<=(Counter const &left, Counter const &right) {
  return !(right < left);
}
```


Speaker notes

For older code, this seems tedious. However, the library Boost.operators will provide the three extra comparison operators automatically, when the less-than operator ist defined.

Like with the equality operators we implement the comparison operators as *hidden friends* to enable symmetry in case implicit conversion is allowed.

Extending Classes : Base

How can we create a `DecrementableCounter`?

```
struct DecrementableCounter : Counter {
  using Counter::Counter; // inherit ctors
  Counter& operator--() & { // --c
    --theCount;
    return *this;
  }
  Counter operator--(int) & { // c--
    Counter result{*this}; // save result
    -- *this; // delegate to prefix
    return result;
  }
};
```


Speaker notes

We can extend a class by “inheriting” from it `struct derived : base {};`:

```
struct DecrementableCounter : Counter {
```

This gives the *derived* class `DecrementableCounter` all the **`public:`** functionality of the *base* class (except the base class `Counter`‘s constructors). To use the base class’ constructors, as if they were constructors of the derived class, one can “inherit” the base class’ constructors with a using declaration:

```
  using Counter::Counter; // inherit ctors
```

However, we cannot access the encapsulated **`private:`** data member, therefore we need to change its accessibility to **`protected:`**

```
struct Counter {
```

```
protected:
  int theCount { };
};
```

Inheriting functionality for Adapters

derived class only has default constructor and not the constructors of its base, unless

```
struct DecrementableCounter : Counter {
  using Counter::Counter; // inherit ctors
```

- **`using`** declaration naming the base's constructor

Speaker notes

Inheriting constructors is a feature introduced with C++11. It makes writing class adapters feasible.

Testing DecrementableCounter

```
void CounterIncrementIncrementsByOne() {
  DecrementableCounter c{42};
  DecrementableCounter const expect{44};
  ++c;
  ASSERT_EQUAL(expect, ++c);
}
void CounterCanPrintItsValue(){
  std::ostringstream out{};
  DecrementableCounter const c { 5 };
  out << c;
  ASSERT_EQUAL("5",out.str());
}
```

```
void
        CounterIncrementDecrementsByOneToZ
         {

  DecrementableCounter c{1};
  ASSERT_EQUAL(DecrementableCounter{1}, c--

  ASSERT_EQUAL(DecrementableCounter{0}, c);

}
void CounterIncrementDecrementsByOne() {
  DecrementableCounter c{42};
  --c;
  ASSERT_EQUAL(DecrementableCounter{40}, --

}
```


Speaker notes

The tests above only show parts of the tests for `DecrementableCounter`.

Pitfalls with Extending by Inheritance

```
struct derived : base {};
class derived : public base {};
```

- derived classes implicitly convert to public bases
  - adding data members can lead to slicing
  - base conversion eliminates features added
- generic code (our testing infrastructure) might not find “hidden friends” of base applied to derived

Speaker notes

While it is possible to have **`private`** base classes, where implicit conversion to the base is suppressed, those are rarely useful and almost the same can be achieved better by a data member of the base class’ type. C++ Advanced and C++ Expert will show

For “specializing” features of the base class, one can prepar the base class member functions with the keyword **`virtual`** to achieve dynamic polymorphism. which we will see later.

A benefit from adapting by inheritance: it provides a new distinguished type that can be used to provide function overloads that won’t accept the base type. Also there are some circumstances where putting an adapted type in a separate namespace together with additional function overloads can achieve desirable results, when the base type’s namespace is not intended to be extended, such as the **`namespace`** `std`.

Exercise 5

https://github.com/PeterSommerlad/CPPCourseIntroductic

Speaker notes

for your own notes

Enumeration Types

```
enum Switch : bool { off, on };
```


Speaker notes

Note that an enumeration using additional names for `bool` is usually not a very good idea….

Defining Enumerations

- For types with only a few named values: enumerators
- unscoped or scoped (**`enum class`**): visibility

```
enum weekday { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
```

- Specify the base type

```
enum weekday : unsigned char { Mon, Tue, Wed, Thu,
    Fri, Sat, Sun };
```

- encapsulate enumerators

```
enum class weekday { Mon, Tue, Wed, Thu, Fri, Sat,
    Sun };
```

- access then only with prefix: `weekday::Wed`

Speaker notes

Either use **`enum class`** for scoped enumerations to avoid spilling all enumerator names into the enclosing scope. Or, put the enumeration declaration within a class type. Then the enumerator names are only visible within the class, or must be prefixed with the class type. I recommend to always specify the base type of an enumeration, that ensures that all values of that base type can be converted to the enumeration type without undefined behavior. The latter might be needed when one defines operator overloads that perform arithmetic on the enumeration values.

Visibility of enumerators

- unscoped enumeration
  - leaks into surrounding scope
  - best as class member

```
namespace calendar {
enum weekday {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};
//Enumerators are visible here
}
bool is_weekend(calendar::weekday day) {
  return day == calendar::Sat ||
         day == calendar::Sun;
}
```

- scoped enumration
  - enumerators must be qualified
  - prefix `weekday::`

```
namespace calendar {
enum class weekday {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};
}
bool is_weekend(calendar::weekday day) {
  return day == calendar::weekday::Sat ||
         day == calendar::weekday::Sun;
}
```


Speaker notes

Like with class types, put enum definitions with the functions operating on them in the same namespace. The above examples deliberately violate this, just for showing the effect.

Providing enumerator Values

- enumerators start with `0`, unless explicitly given

```
enum class month {
  jan=1, feb, mar, apr, may, jun,
  jul, aug, sep, oct, nov, dec
  , january=jan, february, march, april,
  june=jun, july, august, september, october,
  november, december
};
```

- sometimes deliberate “gaps” used for bit-masks:

```
enum class launch_policy : unsigned char {
  sync=1, async=2, gpu=4, process=8, none=0
};
```


Speaker notes

The default value for enumerations is always `0`, even when the initial enum value is set to something else: `month{}` converted to int will have the value `0`.

Defining specific values for enumerators is rare and often found in hardware-specific code.

You can have multiple enumerators with the same value, i.e., `month::jan` and `month::january`

Operator Overloading for enums

- operator overloading possible for enum types

```
namespace calendar {
enum weekday : int { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
weekday operator++(weekday &aday){
  int day = (aday+1)%(Sun+1);
  aday = static_cast<weekday>(day);
  return aday;
}
```

- Put type and operators in a named namespace
- Conversion to and from int required

Speaker notes

Internally enumerations are represented by integer values. That way an enumerator of an unscoped (non-`class`) enumeration can be implicitly converted to an int. For the way back: use `static_cast<weekday>(aday)`

`static_cast<T>(expr)`

- C++ allows explicit conversion between related types
- all numeric types of C++ can convert among each other
  - with potential data loss
- enumerations can convert to/from `int`:

```
int day = (aday+1)%(Sun+1);
aday = static_cast<weekday>(day);
```


Speaker notes

Technically, enumeration values can convert to/from their underlying type (the type specified after the colon `:`). For enum class types without a specific underlying type the type is int (unless you manage to specify enumerators with values outside the range of `int`). For unscoped enumerations without a specified underlying type (legacy/C-style enums), the underlying type is *implementation defined*.

Except for the operator definition of enumerations, where changing the values to and from integers requires explicit conversion with `static_cast<>()` most other applications of explicit type conversion in C++ are a sign of a design mistake.

Note: There is no check that the value converted with `static_cast<weekday>` corresponds to any of the enumerators. For unscoped enumerations without a specific underlying type, this can lead to undefined behavior.💣

There are multiple forms for enforcing type conversion for advanced and encapsulated uses, that I don't show here.

Reminder: postfix++

For postfix increment define

operator++(T &var, int)

```
weekday operator++(weekday &aday,int){
  weekday ret{aday};
  if(aday == Sun) aday=Mon;
  else aday=static_cast<weekday>(1+aday);
  return ret;
}
```


Speaker notes

for your own notes

Output Mapping of Enumerators

- There is no language means to get the spelling of the enumerator values

```
std::ostream& operator<<(std::ostream &out, weekday day){
  constexpr std::array names {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};
  return out << names.at(day);
}
```


Speaker notes

C++26 might bring *reflection* to actually obtain the spellings of enumerator values.

Hide enumeration values in `.cpp` files

```cpp
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_
struct Statemachine {
  Statemachine();
  void processInput(char c);
  bool isDone() const;
private:
  enum State : unsigned short;
  State theState;
};
#endif /* STATEMACHINE_H_ */
```

The *underlying type* allows *forward declaration* of an **`enum`** type.

```cpp
#include "Statemachine.h"
#include <cctype>
enum Statemachine::State: unsigned short {
  begin, middle, end // only usable in .cpp
};
Statemachine::Statemachine() : theState{begin}
        {}
void Statemachine::processInput(char const c) {
  switch(theState){
  case begin :
    if (! isspace(c))
      theState = middle;
    break;
  case middle :
    if (isspace(c))
      theState = end;
    break;
  case end : break;// ignore input
  }
}
bool Statemachine::isDone()const{
  return theState == end;
}
```


Speaker notes

The .cpp file defines the enumeration type of the class `Statemachine::State`. Note that the definition of the enumeration type still needs to use the keyword **`enum`** and all the surrounding required syntax. Only the name must use the class name in its spelling in addition.

Also note that the equality operator **`==`** remains implicitly defined for enum values, as well as the comparison operators. Both take the numeric values into account.

```cpp
static_assert(month::mar==month::march,"enumeration should continue");
```

Exercise 6

exercise06

Speaker notes

We will repeat the exercise 3 exercises, but now with algorithms instead of loops, so you can reuse your test cases.

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise06/

Standard Library Containers

```
#include <vector>
#include <array>
#include <deque>
#include <list>
#include <forward_list>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>
```


Speaker notes

for your own notes

Container flavors in `std::`

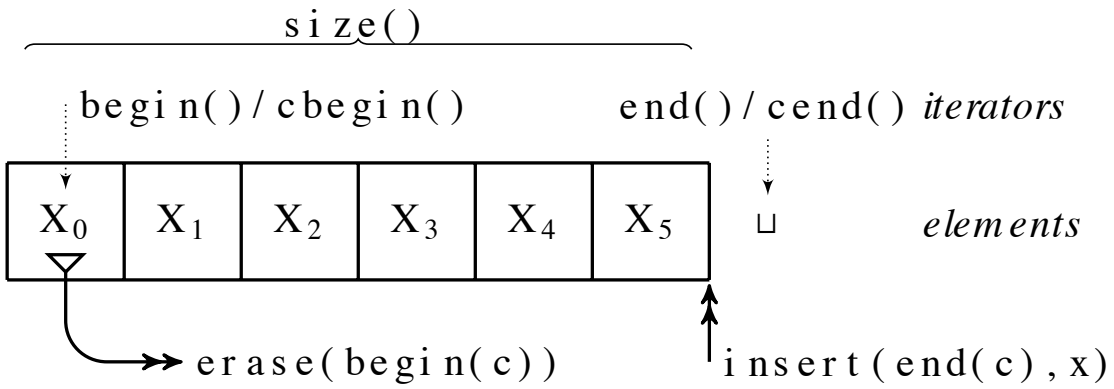
- Sequence Containers
  - ordered access
  - finding element in linear time
- Associative Containers
  - access in sorted order
  - finding elements in logarithmic time
- *Unordered*/Hashed Associative Containers
  - iteration in unknown order
  - finding element in constant time

Speaker notes

Most important container is `std::vector`

The *Unordered* containers are only relevant, when large data sets with frequent element searches and relatively few insertions are used.

Common Container features



- iteration support
- erasing and inserting elements
- `.size()` and `.empty()` queries

Speaker notes

A typical error is to mistake `c.empty()`, that returns if a container `c` is empty or not, with `c.clear()`, which erases all elements from the container `c`.

Common Container API

Containers can be

- default constructed `{}`
  requires specifying element details
- copied from a container of the same type
- *equality-compared* if they are the same type
  or even *lexicographically-compared* with relational operators as long as elements support comparison
- emptied with `clear()`

```
std::vector<int>
          v{};
std::vector
          vv{v};
          //
          copy
if (v == vv){
  v.clear();
}
```


Speaker notes

You might look at

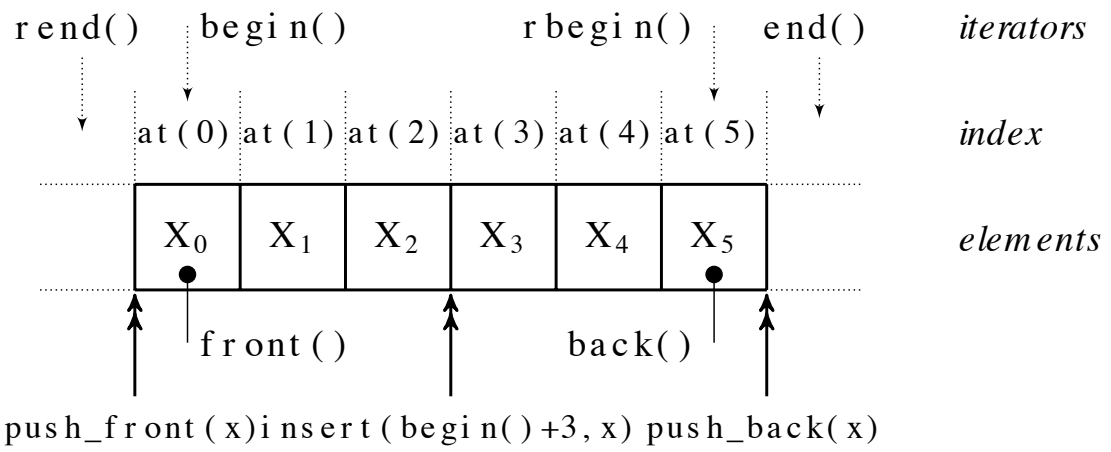
https://en.cppreference.com/w/cpp/container

or

https://hackingcpp.com/cpp/std/containers.html

for further reference

Sequence Containers



order of elements defined by sequence of appending or insertion points

Speaker notes

`std::vector` is the most common and most efficient container.

`std::array` does not require heap allocation, because it has a fixed size and can be used **`constexpr`** code before C++20.

Sequence Container selection

- `std::vector` is the most common and most efficient container.
- `std::array` does not require heap allocation, because it has a fixed size and can be used **`constexpr`** code.
- `std::deque` allows efficient `push_front()` in addition to `push_back()`
- `std::list` is worst choice, only efficient if splicing or insertion in the middle is common
- `std::forward_list` only useful in very special cases, has limited functionality

Speaker notes

`std::string` behaves almost like a std::vector with some special properties.

C++20 makes `std::vector` and `std::string` usable in constexpr contexts, but the compile-time generated objects must not survive until run-time.

Objects of type `std::array` are useful in constexpr contexts and can be created at compile time through constexpr functions and later be used at run-time, for example, to hold fixed parameter values.

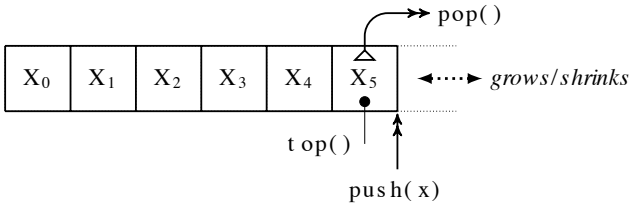
There is no reason to use C-style arrays (which are not explained here) in C++ code (except to implement std::array, for example).

For iterators into containers that become invalid when the container changes, the list containers provide a higher guarantee of validity of such an iterator, i.e., one returned by the find() algorithm, than vector or deque. However, relying on those validity guarantees is error prone and expert-level coding. As an example, if you find an element in at std::list, the iterator of the found element remains valid, even if you insert another element into the list. If you insert a new element into a vector, all iterators to said vector object can become invalid.

Caution `std::vector<bool>` : Unfortunately during standardization of C++, it seemed to be a good idea to provide an efficient bitset implementation through `std::vector<bool>`. This makes it problematic if own generic code must deal with that special case, because `std::vector<bool>` has a different API and behavior than any other `std::vector`. How to deal with that peculiarity will be shown in another course.

LIFO adapter: `std::stack<T>`

```
#include <stack>
```




```
std::stack<int> s{};
s.push(42);
std::cout << s.top();
s.pop();
```

standard containers have a too rich API
`std::stack` limits it to stack operations:
`push()`, `pop()`, `top()` and `empty()`

Speaker notes

`std::stack<T>` adapts `std::deque<T>` and provides just the essential operations of a stack data structure.

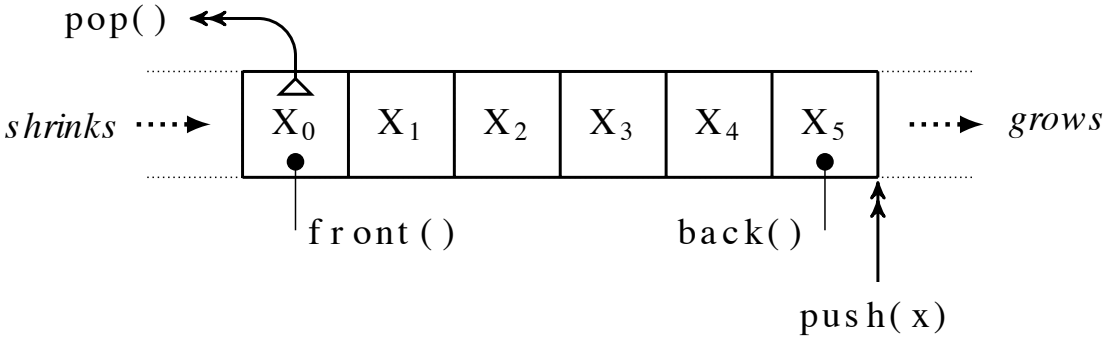
It would also work if parameterized with `std::vector`, but the problem of the special case `std::vector<bool>` was prevented by using std::deque as its default implementation strategy.

It is no longer a container, because it does not allow iteration.

An existing vector `vec` can be adapted to become a `std::stack` as: `auto a_stack { std::stack{vec} };`

Queue Adapter (FIFO)

```
#include <queue>
```



stack vs queue: https://godbolt.org/z/cxEbor4xc

Speaker notes

The `<queue>` header also provides the adapter `std::priority_queue` that pushes elements into a “sorted” order and pops always the smallest element.

Associative (sorted) Containers

- content search, not sequence search
  - search by key
  - access key (set) or key-value pair (map)

	Key	**Key-Value**
Unique Key	`std::set<T>`	`std::map<K, V>`
Multiple Keys	`std::multiset<T>`	`std::multimap<K, V>`

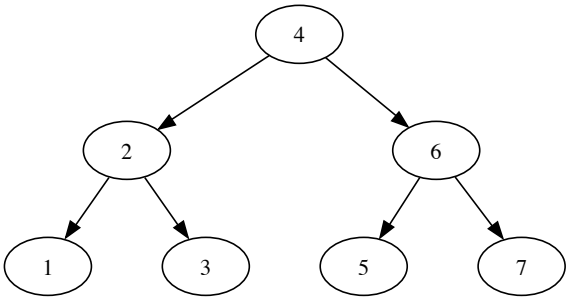
Speaker notes

for your own notes

`std::set` of elements

```
std::set<int> s{7, 1, 4, 3, 2, 5, 6 };
```

- elements in ascending order
  - redefine order possible
- iteration in sorted order
  - keys cannot be modified
- use `.find(k)` `.count(k)`
  - tree search (O(log(n)))
  - `.count(k)` -> `0` or `1`



Speaker notes

Use the member functions for findig and counting instead of the sequential algorithms `std::find()` `std::count()`

C++20 introduces a `.contains(key)` member function. Before that you need to use `.count(key)` to determine if a key is in the set (or map)

Example using `std::set`

```cpp
#include <set>
#include <iostream>
void filtervowels(std::istream &in, std::ostream &out){
  std::set const vowels{'a','e','o','u','i','y'};
  char c{};
  while (in>>c)
    if (! vowels.count(c))
      out << c;
}
int main(){
  filtervowels(std::cin,std::cout);
}
```

https://godbolt.org/z/x5zKdTKxP

Speaker notes

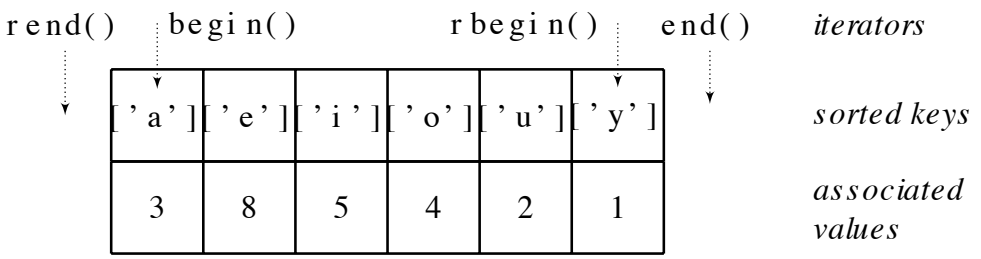
The initializer does not need to be sorted (here it is).

C++20 adds `.contains()`:

https://godbolt.org/z/q5n5j76zj

std::map<key, value>

```
std::map<char,size_t> vowels{{'a',0},{'e',0},{'i',0},{'o',0},{'u',0},
      {'y',0}};
```



https://godbolt.org/z/qasvP76e3

Speaker notes

You must specify `<key,value>` types, for `std::map`, because type deduction from an initializer list of initializer lists doesn't work, because a map's element type is `std::pair<key,value>`

Accessing map elements

map elements are of type ***`std::pair<key,value>`***
generic names ***`.first`*** *and* ***`.second`***

```
for(auto const p : vowels) {
  std::cout << p.first << " = "<< p.second << '\n';
}
```

we can chose better names with a ***structured binding****:*

```
for(auto const [key,value] : vowels) {
  std::cout << key << " = "<< value << '\n';
}
```


Speaker notes

structured bindings to *decompose* `std::pair` and other class types with only public data members can be used for local variables and range-for variables. It does not work for parameter types. The syntax to decompose is **`auto const [ list, of, names ]`**, where the number of elements in the class type it is initialized from must be exactly the number of names listed.

Counting Strings

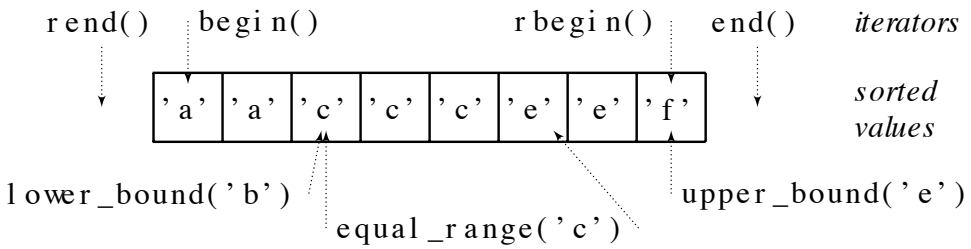
```
void countStrings(std::istream &in, std::ostream &out) {
  std::map<std::string, size_t> occurrences{};
  std::istream_iterator<std::string> inputBegin{in};
  std::istream_iterator<std::string> inputEnd{};
  for_each(inputBegin, inputEnd, [&occurrences](auto const &str) {
    ++occurrences[str]; // creates entry if not existent
  });
  for(auto const & [str,count] : occurrences) {
    out << str << " = " << count << '\n';
  }
}
```

- The indexing **`operator[]`** inserts a new entry if it doesn't already exist.
  - the default value of the "value" type is the new entry's value

Speaker notes

`size_t` is the return type of all `size()` functions and is perfectly suited for counting, because it will only represent unsigned integer values (0 and up). Its range is large, typically bound by 64-bits on modern systems.

Multiset



- `.equal_range()` or `.lower_bound()` / `.upper_bound()` to find sub-ranges of equivalent keys

Speaker notes

Multiset (and multimap) are more tedious to work with, because one needs to accommodate the case of multiple equivalent keys.

Example `std::multiset`

Sorted word list

```
void sortedStringList(std::istream & in, std::ostream & out) {
  using inIter = std::istream_iterator<std::string>;
  using outIter = std::ostream_iterator<std::string>;
  std::multiset<std::string> words{inIter{in}, inIter{}};
  copy(cbegin(words), cend(words), outIter(out, "\n"));
  out<<"---------------------------------\n";
  auto current = cbegin(words);
  while (current != cend(words)) {
    auto endOfRange = words.upper_bound(*current);
    copy(current, endOfRange, outIter{out, ", "});
    out << '\n'; // next range on new line
    current = endOfRange;
  }
}
```

https://godbolt.org/z/Kzrf9Eo73

Speaker notes

The first copy algorithm call will output a single word/string per line. The loop later will put out a line for each unique word, containing all the word occurrences.

https://godbolt.org/z/Kzrf9Eo73

Hashed Containers `unordered_`

- Hashing a key can be a more efficient lookup
  - payoff only when searching for keys is much more frequent than insertion
- no longer sorted by keys
- standard provides `std::hash<T>` only for built-in and standard library types
  - especially for `std::string`
  - no support for `hash_combine()` to ease DIY-hashing for own types
- Creating a good hash-function is non-trivial

Speaker notes

Unfortunately the hashing container support is not perfect. Be aware that frequent insertions can be expensive for unordered containers and can lead to unpredictable performance implications.

If lookups on large structures are really a limiting performance factor of your system, consider unordered_map and unordered_set, otherwise, stick with `std::vector` (linear search is fast for small number of elements due to processor cache locality), `std::set` / `std::map`.

The Boost library provides `boost::hash_combine` to incrementally compute hash values from multiple data members in your type.

std::unordered_set<T>

```cpp
#include <unordered_set>
#include <iostream>
void filtervowels(std::istream &in, std::ostream &out){
  std::unordered_set const vowels{'a','e','o','u','i','y'};
  char c{};
  while (in.get(c)) {
    if (! vowels.count(c)) {
      out << c;
    }
  }
}
int main(){
  filtervowels(std::cin,std::cout);
}
```


Speaker notes

Here we are using `std::istream::get(char)` to retain the white space characters

https://godbolt.org/z/cWjE48n8T

std::unordered_map<K,V>

```cpp
#include<unordered_map>
#include<string>
#include<algorithm>
#include<iterator>
#include<iostream>
void countStrings(std::istream &in, std::ostream &out) {
  std::unordered_map<std::string, size_t> occurrences{};
  std::istream_iterator<std::string> inputBegin{in};
  std::istream_iterator<std::string> inputEnd{};
  for_each(inputBegin, inputEnd, [&occurrences](auto const &str) {
    ++occurrences[str]; // creates entry if not existent
  });
  for(auto const & [str,count] : occurrences) {
    out << str << " = " << count << '\n';
  } // no longer sorted
}
int main(){
  countStrings(std::cin,std::cout);
}
```

- usage like `std::map<K,V>`, except for intrinsic sorting

https://godbolt.org/z/8Yeq1Gxdn

Speaker notes

https://godbolt.org/z/8Yeq1Gxdn

Exercise 7

exercise07

Speaker notes

https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise07/

Effective Algorithms

Speaker notes

Jonathan Boccara's world map of algorithms

`#include <iterator>`

`std::distance(b,e)`
returns number of elements in interval [b,e)

`std::next(it)`
`std::next(it,steps)`
returns copy of it 1/n steps incremented (`++`)

`std::prev(it)`
`std::prev(it,steps)`
copy of it 1/n steps decremented (`--`)

`std::advance(it,steps)`
move it variable by steps, can be negative

Speaker notes

While not officially *algorithms* these functions can be useful when working with iterators.

`std::next` and `std::prev` were introduced with C++11 and are a bit easier to use than `std::advance`.

If the iterators refer to a `std::vector`, `std::array`, `std::string` and `std::deque` these functions are highly efficient. For `std::set`, for example, multiple steps or computing the distance between begin/end of the set requires a loop.

Note that C++20 ranges (not covered here), will provide a range abstraction with lazy evaluation that relies on iterators (and sentinels) underneath.

Iterator/Range concepts

Algorithms require different “strengths” of iterator/ranges.

`copy(b,e,tb)`

`b,e` single-pass single-readable range

`tb` single-pass writable with enough space

`search(b,e,sb,se)`

`b,e` multi-pass, multi-readable

`sb,se` multi-pass, multi-readable range

`sort(b,e)`

`b,e` random-access multiple-read/write access

Speaker notes

Concepts are discovered by the requirements of generic algorithms.

Pre-C++20 the iterators required by the standard library were put into *iterator categories*.

Iterator Categories/Concepts

InputIterator
single pass, single read (`++ *b`)

OutputIterator
single pass, single write (`++ *b=v`)

ForwardIterator
multi pass, multiple read/write(non-`const_iterator`)

BidirectionalIterator
multi-pass with decrement(`--`)

RandomAccessIterator
multi-pass with indexing (`b[i]`)

Speaker notes

C++20 defines an additional concept of “Contiguous Iterator/Range”. This is fulfilled by std::vector, std::array and std::string. All elements are guaranteed to be contiguous in memory. It is important for parallelizes and vectorized versions of the algorithms.

Why Algorithms over loops?

Correctness

It is much easier to call a function correctly than to program a loop

Readability

Applying the correct algorithm expresses intention much better than a loop

Code is read much more often than written (even by the original author)

Performance

Algorithms might perform better than hand-written loops without affecting readability

Speaker notes

There have been measurements of code that was transformed to use algorithms instead of loops, where the converted version was much less source code and did perform better than the original (loopy) code.

https://stroustrup.com/improving_garcia_stroustrup_2015.pdf

Iterators for Ranges

Iterators specify the ranges for algorithms

- `begin(v)` - iterator referring the **first** element in the range `v`
- `end(v)` - iterator referring the position **after the last** element in the range `v`
- if `begin(v)==end(v)` the range has no elements (`distance(b,e) == 0`)

Speaker notes

Before accessing the element an iterator refers to, one needs to guarantee that the iterater actually refers to an element.

Unfortunately, this requires more than one iterator to know if it is valid. And it can become invalid in a way, that is undetectable.

C++20 ranges still use iterators underneath, but provide a safer abstraction by combining a them.

Iterator Adapters for Streams

- `std::ostream` -> `std::ostream_iterator<T>`
- `std::istream` -> `std::istream_iterator<T>`
- `std::istream` -> `std::istreambuf_iterator<char>` - for getting whitespace
  - a `std::istream(buf)_iterator<T>{}` takes the sentinel role of the `end` iterator at the end of the input stream

```
void cat(std::istream & in, std::ostream & out) {
  using in_iter = std::istreambuf_iterator<char>;
  using out_iter = std::ostream_iterator<char>;
  std::copy(in_iter{in}, in_iter{}, out_iter{out});
}
```


Speaker notes

for your own notes

Algorithm `std::for_each`

- execute a function for each element in the range
- function can be given as a function name, lambda, or function object (functor)
  - called for each element with the element as argument
- works on InputIterator
- returns the function object applied (if not parallel)
  - a mutable function object can collect information

Speaker notes

Since C++11 introduced the range-for() statement, `std::for_each()` is less useful. However, std::for_each can be run in parallel with the extra “ExecutionPolicy” parameter, when the underlying range is provided by random-access or contiguous iterators.

A function object is an object of a class type that overloads the function call `operator()(params)` as a member function. Lambdas are technically transformed into such function objects. C++ Insights can show that:

https://cppinsights.io/s/5fd6424b

https://godbolt.org/z/9jrs5zYse

Function object class

A class can overload the function call **`operator()`**

```
struct Accumulator {
  size_t count{0};
  double accumulatedValue{0.0};
  void operator()(double value) & {
    count++;
    accumulatedValue += value;
  }
  auto average() const {
    if (count > 0u)
      return accumulatedValue/count;
    return 0.0; // don't crash
  }
};
```

https://godbolt.org/z/MsfTn1obn

Speaker notes

Such a function object with “memory” can be used in the sequential version of `std::for_each()` that will return the object after it was applied to all elements in the range.

https://godbolt.org/z/MsfTn1obn

Predicate: function returning `bool`

Unary Predicate - one parameter

condition of an element, usually for finding/filtering elements

for algorithms' `_if` version

```
auto is_odd { [](int i){ return i % 2 != 0;} };
```

Binary Predicate - two parameters

usually for comparing elements: `std::sort`

```
auto caselessless { [](char l, char r){
return std::tolower(l) < std::tolower(r);}};
```


Speaker notes

many algorithms take a predicate parameter.

Standard Function Objects

```
#include <functional>
```

Lambdas are simple for simple computations:

```
transform(begin(v),end(v),begin(v),[](auto x){ return -x;});
```

Standard function objects can read better

```
transform(begin(v),end(v),begin(v),std::negate{});
```

Relational operator for sorting (`std::less{}` default)

```
sort(begin(v),end(v),std::greater{});
```


Speaker notes

Binary function objects for arithmetic can be used with the tranform that takes two input ranges.

https://godbolt.org/z/dEh1YvcEa

https://en.cppreference.com/w/cpp/header/functional

Parameterizing Associative Containers

- sorted associative containers can be adapted to change sorting order
  - function object class as additional template argument, default is `std::less<>`
  - binary predicate must be irreflexive and transitive

```
std::set<int,std::greater<int>> reversed_int_set{};
```

- Own function objects can provide special sort order (caseless compare)
  - caution: requirements for sorting must be fulfilled: two values x and y are considered equivalent, when `pred(x,y) or pred(y,x)` is **`false`**

Speaker notes

To see what happens when sorting requirements are not fulfilled try to create a `std::set<double>` and insert `std::sqrt(-2.0)`, this should insert `NaN`. Once a `NaN` value is in the set, no further insertions are possible, because NaN breaks the precondition of std::less. for any number `x < NaN or x > NaN` is false. This means, NaN and x are considered equivalent.

Set for a dictionary

```
using string = std::string; // for brevity
struct caseless{
  bool operator()(string const &l, string const &r) const {
    return std::lexicographical_compare(
      begin(l),end(l),begin(r),end(r),
      [](char lc, char rc) {
        return std::tolower(lc) < std::tolower(rc);
      });
  }
};
using caseless_set = std::multiset<string, caseless>;
```

https://godbolt.org/z/MnP1bcexa

Speaker notes

```
int main() {
  using in = std::istream_iterator<string>;
  caseless_set const wlist{in{std::cin},in{}};
  std::ostream_iterator<string> out{std::cout,"\n"};
  copy(wlist.begin(), wlist.end(), out);
}
```

https://godbolt.org/z/MnP1bcexa

Erase-Remove Idiom

`std::remove()` style algorithms don't remove elements from cotainer but return the iterator, from where to the end the elements can be erased.

```
std::ostream_iterator<unsigned> out{std::cout,", "};
std::vector<unsigned> values{54, 13, 17, 95, 2, 57, 12, 9};
copy(begin(values),end(values),out); std::cout<<'\n';
auto removed = std::remove_if(begin(values), end(values), is_prime);
copy(begin(values),end(values),out); std::cout<<'\n';
values.erase(removed,end(values));
copy(begin(values),end(values),out); std::cout<<'\n';
```


Speaker notes

output:

```
54, 13, 17, 95, 2, 57, 12, 9,
54, 95, 57, 12, 9, 57, 12, 9,
54, 95, 57, 12, 9,
```

Note that after the remove algorithms the remaining elements are moved to the front, but their original copies stay in their original positions.
This is not really guaranteed, you don't know which values are in the to-be-erased positions.

https://godbolt.org/z/q9doEqW4z

`_if` algorithm variations

- some algorithms come with a `_if` suffix, taking a predicate instead of a version searching for a value

```
std::vector<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto nOfPrimes = std::count_if(begin(numbers), end(numbers), is_prime);
copy(begin(numbers),end(numbers),out);
std::cout << "\nhas " << nOfPrimes << " prime nubmers\n";
copy_if(begin(numbers),end(numbers),out,is_prime);
```

count_if	copy_if	replace_if
find_if	remove_if	replace_copy_if
find_if_not	remove_copy_if	

Speaker notes

https://godbolt.org/z/6W8dooPhh

_n algorithm variations

- instead of *end* iterator, provide counter
- must ensure, enough elements are available

```
std::vector<unsigned> numbers{};
generate_n(std::back_inserter(numbers),100,
          [n=1u]()mutable{return n++;});
std::cout << numbers.front() << ".." <<
        numbers.back();
numbers.erase(remove_if(begin(numbers),end(numbers),

        std::not_fn(is_prime)),end(numbers));
std::cout << " has " << numbers.size()
          << " prime numbers\nthe first 10
        are:\n";
copy_n(begin(numbers),10,out);
```

search_n fill_n for_each

copy_n generate_n

Speaker notes

output:

```
1..100 has 26 prime numbers
the first 10 are:
1, 2, 3, 5, 7, 11, 13, 17, 19, 23,
```

https://godbolt.org/z/KG3KzrzYs

Insertion Output Iterators

```
#include <iterator>
```

- Either `.resize(n)` a container accordingly before using it for output, or
- Use an inserter wrapper:

  `back_inserter(c)`
  uses `c.push_back(elt)` on sequence container

  `front_inserter(c)`
  uses `c.push_front(elt)` on `std::deque`

  `inserter(c,it)`
  uses `c.insert(it,elt)` on any container

Speaker notes

for your own notes

Bugs and Pitfalls with algorithms

- Mismatched iterator pairs
  - begin and end from different ranges
- Forget to reserve (enough) space for output range
  - use `(back_)inserter` or `.resize(n)`
- Iterator invalidation
  - e.g. side effect on container within algorithm:

```
std::transform(begin(v),end(v),begin(v),back_inserter(v),std::multiplies{});
```


Speaker notes

I am not showing you wrong code here…

But for example:

```
std::ostream_iterator<unsigned> out{std::cout,", "};
// the following code has undefined behavior!
std::vector<unsigned> v{1,2,3,4,5,6,7,8};
std::vector<unsigned> w{}; // should have been used below:
std::transform(begin(v),end(v),begin(v),back_inserter(v),std::multiplies{});
copy(begin(v),end(v),out);
std::cout<<'\n';
```

“interesting output:” https://godbolt.org/z/1EdrY344r

“address sanitizer can help:” https://godbolt.org/z/hPc6rMfPb

Exercise 8

- exercise08

Speaker notes

- https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise08/

Polymorphism in C++

`auto`

`std::variant`

`virtual`

Speaker notes

polymorph: many forms

polymorphsm in C++ means that a single piece of code behaves differently, depending on the types of its arguments.

static polymorphism
decision what concrete code is executed is made at compile-time, examples are overloading

dynamic polymorphism
the run-time type (dynamic type usually of a sub-class to a base-class interface) defines at run time, which code to execute. Alternative implementation strategy with a fixed set of types uses `std::variant`. *Type-erasure* also allows to use different types in a single place (e.g. `std::function` or `std::any`)

Compile-time Polymorphism

operator overloading and function overloading provide compile-time (static) polymorphism:

```
a + b a * b
```

actual code depends on types of `a` and `b`

- implicit conversions happening
- which operator implementation is used
- for example `std::string::operator+` concatenates

https://godbolt.org/z/o5q4sba6r

Speaker notes

Even for the built-in types there is some kind of “overloading” of operators happening. For example, the machine code for multiplying two `short` variables is different from the code multiplying two `double` variables.

https://godbolt.org/z/o5q4sba6r

Polymorphism with **auto**

```cpp
#include <iostream>
constexpr auto println {
  [](auto what){
    std::cout << what << '\n';
  }
};
int main(){
  println(42);
  println("hello World");
  println(3.14159);
  println(false);
  println(std::hex);
  println(42);
}
```

https://godbolt.org/z/4jYnqdT6e

Speaker notes

https://godbolt.org/z/4jYnqdT6e

C++20 allows the use of **auto** also with normal functions in addition to lambdas. Technically both cases create function templates, which are a topic of C++ Advanced.

```cpp
void println (auto what){
  std::cout << what << '\n';
}
```

Dynamic Polymorphism

Dynamic polymorphism denotes the ability to select a code path at run time, based on the type of an argument.

```
void sayhello(std::ostream &out) {
        out << "!!!Hello World!!!\n";
}
```

- `std::ostream&` reference parameter allows to pass
- a `std::ostringstream` object for our tests
- the `std::cout` object to produce output on stdout
- a `std::ofstream` object to write to a file

*This is achieved through inheritance with member functions declarations using the **`virtual`** keyword*

Speaker notes

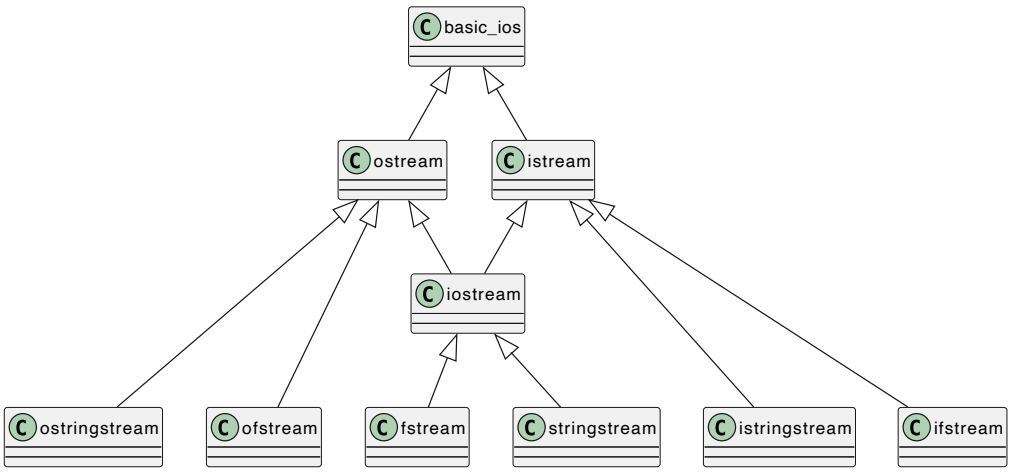
The stream library provides dynamic polymorphism for input and output targets.

In addition it employs static polymorphism by the many overloads of the output **`operator<<`** and input **`operator>>`**

This style of dynamic polymorphism using inheritance and virtual member functions is classically called *object-oriented programming* (OOP)

I/O stream library

`istream` and `ostream` for a hierarchy of types



Speaker notes

This is a simplified representation of the inheritance hierarchy of the iostream classes. See also https://en.cppreference.com/w/cpp/io

To support different character types (*wide* characters to represent non-ascii languages in older C++ versions) the concrete class types for the stream classes have a name prefix of `basic_` and are class templates paramterized by the character type and a feature class (Traits) for dealing with sequences of the character type.

Unfortunately the iostream classes are one of the oldest design parts of C++ and suffer from over- and under-engineering. But because of the backward compatibility principle, cannot be changed. However, the C++ formatting library included in C++20 resolves some of the `ostream` issues (available as `{fmt}` library for C++17.). As of today a corresponding input-consumption equivalent is still missing.

A deep inheritance hierarchy as with the IO stream classes in C++ was considered “cool” in the 1980s/1990s, when object-oriented programming was considered the “best thing since sliced bread”. However, in hindsight, such deep hierarchies and overuse of inheritance can be quite problematic with respect to coupling and ease of change and extension. A typical class hierarchy should be about 2, at most three levels deep, with an abstraction at its root, defining the common API supported by different subclasses implementing/specializing this abstraction in the required different variation.

If dynamic polymorphism is needed, one should decide if unlimited variation is desired (open) or if a fixed number of similar variants is sufficient. In the latter case std::variant might be a simpler option.

For the standard library the openness was beneficial, because I implemented extensions to the stream infrastructure (i.e. for sockets and mmap) and proposed streams that are now standardized (`std::osyncstream` C++20 and `std::spanstream` C++23).

Inheritance Syntax (recap)

- add base class(es) after colon
  - sequence is important

```
class Base {};
class DerivedPrivateBase : Base {};
struct DerivedPublicBase : Base ();
```

- Interface inheritance requires public Base
  - private inheritance is possible for mix-in classes providing hidden friends

```
class Base {};
struct MixIn {};
struct MultipleBases : public Base, private MixIn ();
```


Speaker notes

for your own notes

Initialization with multiple Bases

- Base class constructors can be explicitly called like member initializers
  - if omitted, default construction of Base
- Base classes constructed before Derived's members
  - put Base ctor call first

```
class DerivedWithCtor : public Base1, public Base2 {
  int mvar;
public:
  DerivedWithCtor(int i, int j) :
      Base1{i}, Base2{}, mvar{j} {}
};
```


Speaker notes

Implicit reordering of Base class constructors and Derived class data members happens in the sequence of their definitions, regardless of the order given in the constructor's initialization list. So keep those in sync.

What is the output?

```
struct Base1 {
  explicit Base1(int value) {
    std::cout << "Base1 with argument " << value << "\n";
  }
};
struct Base2 {
  Base2() { std::cout << "Base2\n"; }
};
class DerivedWithCtor : public Base1, public Base2 {
  int mvar;
public:
  DerivedWithCtor(int i, int j)
    : mvar{j}, Base2{}, Base1{mvar} {
      std::cout << "Derived with mvar:" << mvar <<'\n';
    }
};
int main() {
  DerivedWithCtor dwc{1, 2};
}
```

https://godbolt.org/z/4E6rKf8e8

Speaker notes

https://godbolt.org/z/4E6rKf8e8

Due to the wrong order, `Base1{mvar}` uses mvar before it is initialized. this can lead to undefined behavior or at least surprising behavior.

Dynamic Polymorphism with Inheritance

- C++' default mechanisms support value classes with copying/moving and deterministic lifetime
- Operator and function overloading and templates allow polymorphic behavior at compile time
  - This is often more efficient and avoids indirection at run-time
- Dynamic polymorphism needs object references or (smart) pointers to work
  - Syntax overhead
  - The base interface must be a good abstraction
  - Copying carries the danger of slicing (an object is only copied partially)
    - Therefore prevent all copying of classes from a class hierarchy
- Implementing design patterns for run-time flexibility: i.e., Strategy, Composite, Decorator
  - Client code uses abstract interface and gets parameterized/called with reference to concrete instance
- But: if run-time flexibility is not required, templates can implement many patterns with compile-time flexibility as well

Speaker notes

for your own notes

Example: pacman board elements

```
struct Wall{
  bool canPass() const { return false; }
};
struct Empty{
  bool canPass() const { return true;  }
};
struct Pellet{
  bool canPass() const { return true; }
  void consumed(Pacman &p) { p.eat(*this); }
};
struct PowerPellet{
  bool canPass() const { return true; }
    void powerup(Pacman&p) { p.eat(*this);  }
};
struct Door{
  bool canPass() const { return true; }
  void passThrough(Pacman&p) { p.warp(*this);  }
};
```


Speaker notes

Five different Elements of the Board. Each can tell if pacman can pass through it.

Three of them have a specific side effect of pacman walking into it.

Variation with std::variant

```
using BoardElement=std::variant<Empty,Wall,Door,Pellet,PowerPellet>;
bool canPass(BoardElement be){
  return std::visit([](auto elt){
    return elt.canPass();
  }, be); // dynamic delegation to type
}
void passElement(Pacman &pacman, BoardElement be){
  std::visit(overloaded{
    [](auto){FAILM("not a visited element");},
    //[](Wall){ throw std::logic_error{"cannot pass through walls"} ;},
    [](Empty){/* nothing */},
    [&pacman](Door d){ d.passThrough(pacman);},
    [&pacman](Pellet p){ p.consumed(pacman);},
    [&pacman](PowerPellet p){ p.powerup(pacman);}
  },be); // type-based "switch"
}
```


Speaker notes

`std::visit` requires as its first argument a function object that can accept all member types as argument.

The generic lambda with auto fulfils that for canPass, because all types implement canPass.

For the passElement function we use a “trick” of combining different lambdas into a single function object that delegates to the matching one accordingly. We either need to list all possible types as function arguments, or have another generic one that is used when neither of the given ones fails to accept it. If not, we get a compile error. So in case we have an “auto” parameter function listed with “`overloaded`”, it is best, if that one raises a run-time error. Otherwise, required changes, when the variant is extended might be missed.

Dynamic Polymorphism Base

```
struct BoardElement {
  virtual bool canPass() const =0;
  virtual void passElement(Pacman &) &{}
  virtual ~BoardElement() = default;
  BoardElement& operator=(BoardElement&&) & noexcept = delete;
  // Rule of DesDeMova prevents slicing
};
```

- mark member functions **`virtual`**
- not implemented defined as **`=0;`**

Speaker notes

We have to rewrite our class types to conform to a common API and put that API into the base class.

Member functions, that we want all subclasses to implement can be defined as *pure virtual* by defining them as `=0;` This also prevents the class from being used to create any object.

If derived classes are created on the heap (C++ Advanced) and then released via a base class pointer, we need to mark the destructor as virtual and define it with `=default;`. This further implies that we want to prevent the base class to allow copies. The least code to achieve this, is to define the move assignment operator as `=delete;`. This trick prevents all automatically provided copy and move operations on the base class and derived class objects.

Constructing a board of these board Elements now would need to use an indirect means, so we no longer can just use values to store different subtypes of board elements in the board.

Dynamic Polymorphism Derived

```
struct Wall : BoardElement {
  bool canPass() const { return false; }
  void passElement(Pacman &) & {
    throw std::logic_error{"cannot pass walls"};
  }
};
struct Pellet : BoardElement {
  bool canPass() const { return true; }
  void passElement(Pacman &p) & { p.eat(*this);
        }
};
```

```
struct Empty : BoardElement {
  bool canPass() const { return true;  }
};
struct PowerPellet : BoardElement {
  bool canPass() const { return true; }
  void passElement(Pacman&p) & { p.eat(*this);  }
};
struct Door : BoardElement {
  bool canPass() const { return true; }
  void passElement(Pacman&p) & { p.warp(*this);
        }
};
```


Speaker notes

Each derived class for each of the elements needs to implement the function `canPass() const` that is a pure-virtual function in the base. Note that the signature in the derived class must match exactly.

Those that need to do something special when passing, needs to implement `passElement() &` member function.

Employing dynamic Polymorphism

To actually make use of the language virtual dispatch, one needs to pass the objects by base-class reference

```
bool canPass(BoardElement const &be){
  return be.canPass();
}
void passElement(Pacman &pacman, BoardElement &be){
  be.passElement(pacman);
}
```


Speaker notes

Compare the use of `BoardElement&` with our uses of `std::ostream&`. If only const-member functions are called, pass by const-reference is sufficient.

Pitfalls with Inheritance

- Shadowing
- Signatures that are off
- Need to use Indirection
- Copy Prevention

Speaker notes

for your own notes

Shadowing

- derived class member with same name shadows base
- but won't be called through base, unless virtual

```
struct Base {
  void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};
struct Derived : Base {
  void sayHello() const {
    std::cout << "Hi, I'm
        Derived\n";
  }
};
```

```
void greet(Base const & base) {
  base.sayHello();
}
int main() {
  Derived derived{};
  greet(derived);
}
```


Speaker notes

https://godbolt.org/z/63bs6E1TE

try with virtual:

https://godbolt.org/z/ohn1a6fhM

Signatures when Overriding

```
struct Base {
  virtual void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};

struct Derived : Base {
  void sayHello() override { // ⚡
    std::cout << "Hi, I'm Derived\n";
  }
};

struct OtherDerived : Base { // ⬇️⚡
  void sayHello(std::string name) const override {
    std::cout << "Hi " << name << ", I'm OtherDerived\n";
  }
};
```

- signatures must match, same name hides base names

Speaker notes

Try it for effects:

https://godbolt.org/z/4Mhjd8een

To add an overload with the same name but other signatures in a derived class but while retaining the base's member functions use a using declaration to import the names:

```
struct OtherDerived : Base {
  using Base::sayHello; // obtain base member function
  void sayHello(std::string name) const override {
    std::cout << "Hi " << name << ", I'm OtherDerived\n";
  }
};
```

Need for References

```
struct Base {
  void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};
struct Derived : Base {
  void sayHello() const {
    std::cout << "Hi, I'm Derived\n";
  }
};
void greet(Base const base) {
  base.sayHello(); // always base
}
```

```
struct Base {
  virtual void sayHello() const {
    std::cout << "Hi, I'm Base\n";
  }
};
struct Derived : Base {
  void sayHello() const {
    std::cout << "Hi, I'm Derived\n";
  }
};
void greet(Base const &base) {
  base.sayHello();
}
```


Speaker notes

Play with it:

https://godbolt.org/z/95sdjPfvn

https://godbolt.org/z/Kxd8cjzPh

Preventing Copy in Base

virtual in Base means interface, prevent slicing

- pure virtual member prevents copy

```
struct BoardElement {
  virtual bool canPass() const =0;
  virtual ~BoardElement() = default;
};
```

- Rule of DesDeMovA

```
struct BoardElement {
virtual void passElement(Pacman
        &) & {}
virtual ~BoardElement() =
        default;
BoardElement&
operator=(BoardElement&&) & =
        delete;
};
```

one virtual means you need a virtual destructor

Speaker notes

I recommend that you always apply the Rule of DesDeMovA when you define a base class with a virtual member function and a virtual destructor.

While technically it is possible to omit the virtual destructor in the base class, when you never ever allocate an object of derived class on the heap an deallocate it via a base class pointer, I do not recommend it, because it means any change to the system can break this.

Unfortunately `std::unique_ptr` does not prevent derived-to-base conversion without a virtual destructor in the base (it would be possible to detect that!)

Many more details and guidelines in C++ Advanced!

Outlook Inheritance and Classes

- C++ Advanced will provide much more on good class design.
- try to stick with value types and use virtual only, when you know your abstraction will be extended in the future and the base unchanged

Inheritance couples classes very tightly, remember you cannot get rid of your genes!

Speaker notes

for your own notes

Exercise 9

exercise09

Speaker notes

- https://github.com/PeterSommerlad/CPPCourseIntroduction/blob/main/exercises/exercise09/

Done...

Feel free to contact me @PeterSommerlad or peter.cpp@sommerlad.ch in case of further questions

Speaker notes

for your own notes