C++ Advanced

© 2022 Peter Sommerlad

C++ Advanced

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

Slides:

https://github.com/PeterSommerlad/CPPCourseAdvanced/tree/main/

© 2022 Peter Sommerlad

My philosophy

Less Code

=

More Software

© 2022 Peter Sommerlad

Speaker notes

I borrowed this philosophy from Kevlin Henney.

© 2022 Peter Sommerlad

What is in C++

© 2022 Peter Sommerlad

Speaker notes

In the C++ Introduction we look solely on parts of C++ that are relevant for App development.

This course we start to look at C++ parts that cross over to the library language, but mostly to cover things that might have been used in existing applications without need (or that are no longer needed in more recent C++ versions). C++ Expert will dive more deeply into the “dirty” language features that might be needed for library code, but that requires careful attention to detail to not be misused.

© 2022 Peter Sommerlad

In this course

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

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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++98
- Other C++ IDEs (vscode, clion)

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

C++ Resources

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

© 2022 Peter Sommerlad

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/CPPCourseAdvanced

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

C++ Introduction Recap

- `auto const var{init};` - almost always auto
- non-const variables only for side-effects and return values
- unit testing with CUTE
- prefer algorithms over loops (body is lambda)
- pass by value or pass by reference (**&**)
- member functions with side effects have **&** qualifier
- operator overloading (C++20 comparison)

© 2022 Peter Sommerlad

Speaker notes

this is just a very brief list of things, that we covered, for those who did not participate in my C++ Introduction course and only those things that we neither repeat here nor that are always common practice, even if they should be.

© 2022 Peter Sommerlad

Functions

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


© 2022 Peter Sommerlad

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`.

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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?

© 2022 Peter Sommerlad

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*

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

Function Parameter Kind

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.

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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`
		`}`

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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

Functions are first class objects

```
#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
}
```


© 2022 Peter Sommerlad

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:

```
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.

© 2022 Peter Sommerlad

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);
}
```


© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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);
}
```


© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

Speaker notes

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

© 2022 Peter Sommerlad

-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

© 2022 Peter Sommerlad

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.)

© 2022 Peter Sommerlad

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()`

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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;
}
```


© 2022 Peter Sommerlad

Speaker notes

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

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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'; }
}
```


© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

Speaker notes

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

© 2022 Peter Sommerlad

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);
  }
}
```


© 2022 Peter Sommerlad

Speaker notes

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

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

© 2022 Peter Sommerlad

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;
}
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

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)

© 2022 Peter Sommerlad

4.a `std::terminate()`

*often **terminate** is the right reaction*

Covering up errors for too long can hide problems

Throwing from a **`noexcept`** functions calls `std::terminate()`

Might not be viable in production.

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Wide Contract Functions cannot fail

Simplest to use, when nothing can go wrong

no external dependencies

Postcondition violation only happens with external dependencies

no preconditions

Easiest to use, caller cannot be wrong.

`noexcept` marks wide-contract functions.

© 2022 Peter Sommerlad

Speaker notes

Wide Contract functions accept all possible values/value combinations from their parameter types

© 2022 Peter Sommerlad

Narrow Contract Functions can fail

have run-time preconditions

and should be `noexcept(false)`,
unless `terminate()` is a viable option

Failure mode: ***programmer error***

© 2022 Peter Sommerlad

Speaker notes

Narrow Contracts have run-time preconditions

If a narrow contract is violated it might be best to std::terminate,

because the software must be fixed, unless termination is an abomination.

© 2022 Peter Sommerlad

Functions with external dependencies

- memory allocation
- I/O
- resource allocation

can lead to postcondition violation

Failure mode: systen or human

© 2022 Peter Sommerlad

Speaker notes

If failure due to external dependencies can be expected and circumvented, deal with it.

If failure cannot be fixed and hinders continuation: terminate/abort, i.e., out of memory.

© 2022 Peter Sommerlad

Error taxonomy (after Herb Sutter)

	What to use	Report-to	Handler
abstract machine corrupt (UB, stack)	`terminate()`	user	human
bug, precondition violation	asserts, contracts could abort	Programmer	human, write better tests
recoverable, expected error	throw exception, return error	calling code	code for fault tolerance

available: youtube link

© 2022 Peter Sommerlad

Speaker notes

Herb Sutter's talk:

https://www.youtube.com/watch?v=ARYP83yNAWk

© 2022 Peter Sommerlad

Error handling strategy

Have a conscious error handling strategy !

- understand its limitations
- strategy does not need to be uniform
- which ist best? … **it depends**

Distinguish functions by their need for error reporting !

*Violating preconditions is **always** a programmer error.*

Unit test error situations

© 2022 Peter Sommerlad

Speaker notes

Do not shoehorn a single mechanism over all possible error conditions. This can lead to hard-to-use systems.

Nevertheless, strive for similarity in error reporting and handling within a subsystem and on its boundary.

See Expert course for effective mapping of exceptions to error codes and vice versa.

© 2022 Peter Sommerlad

Exercise 1

exercises/exercise01.md

© 2022 Peter Sommerlad

Speaker notes

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise01.md

© 2022 Peter Sommerlad

Function Templates

© 2022 Peter Sommerlad

Speaker notes

- function templates
- template parameter concepts
- C++20 concepts
- generic lambdas
- variadic templates and lambdas
- fold expressions

© 2022 Peter Sommerlad

What you already know

```
[x=1](auto y) mutable {return (x++) * y ; }
```

- lambdas allow **`auto`** as parameter type
- capture variable type is deduced
- return type is deduced

All types are deduced from context

© 2022 Peter Sommerlad

Speaker notes

The calling argument defined the type used for the parameter.

The context and definition of the capture variable defines its type. The keyword **`mutable`** allows to manipulate it.

The return statement(s), if any, define the return type of the lambda. This return-type deduction is also available for inline functions.

© 2022 Peter Sommerlad

Origin of `template`

- type-safe containers (`std::vector<T>`)
  - type-parameterized types and functions
  - no copy-paste of code and no macros
- compile-time polymorphism
- reuse of code not yet written
  - `std::vector` exists before `class MyClass` still `std::vector<MyClass>` works

But before we look at function templates.

© 2022 Peter Sommerlad

Speaker notes

While types are the most common template parameters, one can also use class templates as template parameters and values as template parameters. Before C++20 template value parameters must be of integral type (plus some extras), with C++20 almost all compile-time types (literal types) that are comparable with strong-equality are usable. In addition the type of the a value template paramter can be deduced with `auto` in C++20 from the concrete argument.

© 2022 Peter Sommerlad

Motivation function templates

```
namespace MyMin{
inline int min(int a, int b){
    return (a < b)? a : b ;
}
inline double min(double a, double b){
    return (a < b)? a : b ;
}
inline std::string min(std::string a, std::string b){
    return (a < b)? a : b ;
}
}
```

- multiple overloads with identical bodies
- not easily extendible for new types
- common header implies dependency to **<string>**
- ambiguities if arguments don't match

© 2022 Peter Sommerlad

Speaker notes

implicit conversions or mismatched parameter types can lead to ambiguities

Compiler explorer: https://compiler-explorer.com/z/6s35o95jo

```
#include <iostream>
#include <string>

namespace MyMin{
inline int min(int a, int b){
    return (a < b)? a : b ;
}
inline double min(double a, double b){
    return (a < b)? a : b ;
}}
namespace MyMin{
inline std::string min(std::string a, std::string b){
    return (a < b)? a : b ;
}
}
int main(){
    using MyMin::min;
    std::cout << "min(1,2) : " << min(1,2) << '\n';
    std::cout << "min(1.,2.) : " << min(1.,2.) << '\n';
    std::cout << "min(one,two) : " << min("one","two") << '\n';
//    std::cout << "min(1,1.1) : " << min(1,1.1) << '\n'; // ambiguous
}
```

© 2022 Peter Sommerlad

Function Template for `min()`

```
#ifndef MYMIN_H_
#define MYMIN_H_
namespace MyMin{
template <typename T>
T min(T a, T b){
        return (a < b)? a : b ;
}
}
#endif
```

- **`template`** keyword with
  - **`<>`** for template parameters
  - **`typename`** for type parameters
  - function definition
- definition in header file
  - implicitly **`inline`**

© 2022 Peter Sommerlad

Speaker notes

Function templates are implicitly inline functions, even without the keyword **`inline`**

Template parameters in **`<>`** can be

- types (**`typename`** or **`class`**),
- class templates (**`template`**), or
- values/NTTP (*integral* type or **`auto`** (C++20))
  - in addition to integral types, pointers and references are also possible
  - C++20 extends the types for values to “structural types” as well, which can be structs(literal types) with public members and bases or arrays of structural types. Creating a special structural type, C++20 even allows string literals to be used as template arguments. Type deduction using **`auto`** for NTTPs was introduced in C++20.

Often it makes sense to define function templates as **`constexpr`** functions, to allow their use at compile time.

Using value template parameters is an advanced topic, typically employed in template-meta-programming. Before C++20, value parameters where called “non-type template parameters” **NTTP**. They

© 2022 Peter Sommerlad

Function Template usage

```cpp
#include "MyMin.h"
#include <iostream>
#include <string>

int main(){
  using MyMin::min;
  using namespace std::string_literals;
  std::cout << "min(1,2) : " << min(1,2) << '\n';
  std::cout << "min(1.,2.) : " << min(1.,2.) << '\n';
  std::cout << "min(one,two) : " << min("one","two") << '\n'; // accidental!
  std::cout << "min(three,two) : " << min("three","two") << '\n'; // ??
  std::cout << "min(three,two) : " << min<std::string>("three","two") << '\n'; // ??
  std::cout << "min(three,two) : " << MyMin::min("three"s,"two"s) << '\n'; // ??
//    std::cout << "min(1,1.1) : " << min(1,1.1) << '\n'; // ambiguous
}
```

you cannot directly compare “literals”!

https://compiler-explorer.com/z/ET3d45j35

© 2022 Peter Sommerlad

Speaker notes

https://compiler-explorer.com/z/ET3d45j35

© 2022 Peter Sommerlad

Template Instantiation

- called name -> function template
  - `min(1,2)` -> `template min`
- function argument types -> template arguments
  - `1` - `int`, `2`-`int` -> `min<int>`
- **instantiation** with template arguments
  - `int min<int>(int a, int b){...}`
- type checking of body with concrete arguments
  - can fail with “interesting” messages

observe template instantiation with cppinsights:

https://cppinsights.io/s/6dfe51af

© 2022 Peter Sommerlad

Speaker notes

Type checking for templates happens twice: * first, when the template itself is processed by the compiler, but only coarsely, because information on the template arguments is missing and * then a second time, when the template is instantiated, with the concrete template arguments deduced or given.

Compile failures often happen at instantiation time:

- template argument cannot be deduced, because of ambiguities
- template argument does not fulfil the requirements of the function template body (concept not matched)

Sometimes the error is in the actual template and not where the error is detected by the compiler

observe template instantiation with cppinsights:

https://cppinsights.io/s/6dfe51af

© 2022 Peter Sommerlad

Template *Magic*

Template Argument Deduction

- Function argument types deduce actual template typename arguments
- If ambiguous -> compile error

```
std::cout << "min(1,1.1) : " << min(1,1.1) << '\n';
```

Explicit instantiation helps

```
std::cout << "min(1,1.1) : " << min<double>(1,1.1) << '\n';
```


© 2022 Peter Sommerlad

Speaker notes

One of the *magic* parts of C++ is the **template argument deduction** from the function arguments of a call of a function template.

If that magic fails, for example due to ambiguities or the intended implicit conversion, one can specify the template arguments explicitly by using a template id (`min<std::string>`) instead of just the template name (`min`) for calling the function template:

```
min<std::string>("hello","Peter")
```

© 2022 Peter Sommerlad

aside: type of `"literal"`

string `"literals"` have the type `char[N+1]`

- parameter passing decays it to *pointer to const char*
  - size information is lost
  - use: `std::array`, `std::string`, or `std::vector`

pass `"literals"s` with a suffix (`""s` or `""sv`):

```
using namespace std::literals;
min("peter"sv, "toni"sv) // std::string_view
```

or specify `<std::string>` a template argument when possible.

```
min<std::string>("peter", "toni")
```


© 2022 Peter Sommerlad

Speaker notes

The effect of treating string literals as array of `char` with an array-length including the implicit `'\0'` C-style string termination character is a legacy, we cannot ignore, but which should not dominate your use of strings when programming C++. To interface with C-library functions expecting a string, the `std::string` class provides the member function `.c_str()` that will provide the corresponding conversion to a *pointer to const char* which in turn points to the internals of the `std::string` object.

We won't use plain (C-style) arrays or pointers in modern C++, except tightly encapsulated, because their use is error prone and their types support too many operations that can easily lead to incorrect or undefined behavior.

© 2022 Peter Sommerlad

Template Parameter Concepts

```
template <typename T>
T min(T a, T b){
    return (a < b)? a : b ;
}
```

- Requirements on template parameter are implied by its usage
- Such implicit requirements are also called “concept” of the template parameter

What is the concept of type T?

© 2022 Peter Sommerlad

Speaker notes

The concept of type `T` in `min()`:

- T must support pass/return by value (= copying)
- Values of type T must be comparable with `operator<(T,T)`
  - and the comparison must return a value convertible to `bool`, because of its use as the condition in the conditional operator (`?:`)
    - e.g. `void operator<(T,T)` would be invalid.

If the template argument doesn’t match the concept of the template parameter you get an error message, often in the template code itself, even if the reason is at the site of the template instantiation

```
#include <iostream>
#include <string>
namespace MyMin{
template<typename T>
T min(T a, T b){
    return (a < b)? a : b ; // <-- error here for missing operator< for X
}
}
struct X{
    int i;
}; // cannot compare
int main(){
    using MyMin::min;
    std::cout << "min(1,2) : " << min(1,2) << '\n';
    std::cout << "min(X{2},X{1}) :" << min(X{2},X{1}).i << '\n'; // error cause here
}
```

https://compiler-explorer.com/z/5axP9Pvn3

© 2022 Peter Sommerlad

Quiz Parameter Concepts

```
template <typename T>
T const & max(T const & a, T const & b){
    return (a > b)? a : b ;
}
```

What is the concept of `T` for `max()`?

Why is it OK to return a reference here?

What happens with arguments of different types?

© 2022 Peter Sommerlad

Speaker notes

The concept of type `T` in `max()`:

- Values of type `T` must be comparable with `operator<(T,T)`
  - and the comparison must return a value convertible to `bool`, because of its use as the condition in the conditional operator (`?:`)
    - e.g. `void operator<(T,T)` would be invalid.
- anything else?

© 2022 Peter Sommerlad

Specifying concepts for Template Parameter (C++20)

concept *to name template parameter requirements*

```
template <typename T>
concept theConcept = requires ... ;
```

requires *to specify such requirements*

```
... = requires requires (T a) { a < a };
```


© 2022 Peter Sommerlad

Speaker notes

Unfortunately, the parser of the Cevelop IDE does not support C++20 syntax (yet), so while you can still work with code specifying template concepts, the IDE might mark the corresponding source code as erroneous, even if it isn't. Don't let yourself confuse by that.

And yes, **`requires requires`** is correct. The first one specifies that there is a requirement, the second ome forma a *require expression* that specifies what the actual requirement is. When using a named concept as a requirement, only one `require` and the concept id is required (pun intended!).

We will see further examples of *require expressions* later.

However, we will not use C++ concepts for specifying templates in most of the course, so that you knowledge also fits with older C++ versions and code.

© 2022 Peter Sommerlad

Specifying concept for `min()`

```
template <typename T>
requires requires (T a) {
  a < a ;
}
T min(T l, T r) {
  return l < r ? l : r;
}
```

```
#include <concept>
template <typename T>
requires requires (T a) {
  { a < a } // extra {}
    -> std::convertible_to<bool>;
}
T min(T l, T r) { // ...
```

- **`operator<`** must work for values of T

  or better:

- and it returns a type convertible to `bool`

© 2022 Peter Sommerlad

Speaker notes

A require expression can show C++ syntax examples that must be valid for the template parameter or values of it, if it is a typename parameter.

There are several syntax options for such a specification. This first example shows a direct constraint.

`std::convertible_to<T>` is a concept provided by the standard library.

Alternatively, the concept `std::same_as<bool>` could be used to insist on having exactly `bool` return type.

after the `->` one must use a concept and cannot specify the type of the result directly. Here, a concept takes one less template argument than its parameter list demands, because result type is implicitly used as the first argument of the concept. That means, the definitions of the concepts `std::convertible_to` and `std::same_as` actually have two template typename parameters.

© 2022 Peter Sommerlad

Naming concept for `min()`

```
template <typename T>
concept LessThanComparable = requires (T a) {
    { a < a } -> std::convertible_to<bool>;
};
```

- concept definitions are templates
- their “value” must be a compile-time `bool` expression
  - requires expression
  - other concepts in logical conjunction/disjunction
  - `bool` variable templates
- named concepts can be reused

© 2022 Peter Sommerlad

Speaker notes

Named concepts can have one or more template parameters.

Note that in some contexts the first of the template arguments is implicit, e.g., after the `->` in a requires expression

© 2022 Peter Sommerlad

Using a named concept for `max()`

as requirement:

```
template <typename T>
requires LessThanComparable<T>
T max(T l, T r) {
  return !(l < r) ? l : r;
}
```

as **`typename`** replacement, implicit requirement

```
template <LessThanComparable T>
T max_short(T l, T r) {
  return !(l < r) ? l : r;
}
```


© 2022 Peter Sommerlad

Speaker notes

There are several syntactical options for referring to named concepts (unfortunately).

One can use a requires clause, or directly use the concept name in place of **`typename`**

A further option is to use the “shorthand” notation:

```
auto max_shortest(LessThanComparable auto l, LessThanComparable auto r)
requires std::same_as<decltype(l),decltype(r)>
{
  return !(l < r) ? l : r;
}
```

This last version looks almost like a generic lambda with additional use of the concept name. However, there is no way to specify that the deduced type of the two arguments is the same, unless one uses an additional requires clause. This uses the compile-time “reflection mechanism” **`decltype(var)`** to determine the actual type deduced for the function parameter. As one can see, when parameters of a function template need to have a common type, it is best to actually spell out the template prefix.

© 2022 Peter Sommerlad

cross-C++-version compatibilty

C++ defines feature-test macro: `__cpp_concepts`

```
#ifdef __cpp_concepts
#include <concepts>
#endif

template <typename T>
#ifdef __cpp_concepts
requires requires (T a) {
  { a < a } -> std::convertible_to<bool>;
}
#endif
T min(T left, T right) {
  return left < right ? left : right;
}
```


© 2022 Peter Sommerlad

Speaker notes

Feature-test macros are available for many features since C++11. They allow to employ the preprocessor's conditional compilation to enable/disable lines of the source code.

Using the longer require syntax for templates allows to create code that states concepts in pre-C++20 compilable way.

The above example compiles with C++17, but will just give you a different error message, when the concept is not satisfied.

https://godbolt.org/z/h6sfcsYGz

© 2022 Peter Sommerlad

Template Alternative Lambdas

```
auto const min{ [](auto a, auto b){return a < b ? a : b;}};
```

supports heterogenous arguments

```
cout << "min(1,2) : " << min(1,2) << '\n';
cout << "min(1.1,2) : " << min(1.1,2) << '\n';
cout << "min(one,two) : " << min("one","two") << '\n'; // accidental!
cout << "min(three,two) : " << min("three","two") << '\n'; // ??
cout << "min(three,two) : " << min("three"s,"two"s) << '\n'; // works
cout << "min(1,1.1) : " << min(1,1.1) << '\n';
```

https://compiler-explorer.com/z/jonE7bscn

© 2022 Peter Sommerlad

Speaker notes

https://compiler-explorer.com/z/jonE7bscn

internally

https://cppinsights.io/s/d400e3aa

As with the shorthand notion of function templates in C++20, one can prefix the `auto` parameter declaration with a named concept to specify a constraint for the lambda function parameter.

© 2022 Peter Sommerlad

Generic Lambdas and Function Templates

```
auto const min{ [](auto a, auto b){return a < b ? a : b;}};
```

internally becomes:

```
class __lambda_3_17
{
  public:
  template<typename T0 typename T1>
  inline constexpr auto operator()(T0 a, T1 b) const
  {
    return a < b ? a : b;
  }
};
__lambda_3_17 const min {__lambda_3_17{}};
```


© 2022 Peter Sommerlad

Speaker notes

Generated with and simplified from https://cppinsights.io/s/3c7226e3

© 2022 Peter Sommerlad

Problems with Function Templates

```
std::cout << MyMin::min("hello","world") << '\n';
```

may not work as expected, because of pointer decay

```
template<typename T>
T const & max(T const & left, T const &right) {
```

fails to compile if string literals have different lengths

```
<source>:40:26: error: no matching function for call to 'max(const char
[6], const char [5])'
   40 |   std::cout << MyMin::max("short","long") << '\n'
```


© 2022 Peter Sommerlad

Speaker notes

no overloading for lambdas possible (sometimes very good!)

Overloading allows you to specify versions of your function template for specific cases, eg. when being used with string literals.

© 2022 Peter Sommerlad

Special overload for string literals (1)

```
auto min(char const * left, char const * right) {
  return std::string_view(left) < std::string_view(right) ? left : right;
}
```

Because of pointer decay of `"hello"` this overload matches

https://godbolt.org/z/vjY8Wc35r

© 2022 Peter Sommerlad

Speaker notes

Alternatively, one can even directly match the underlying character array by reference and a template. however, in C++17 having two function template overloads that match causes an ambiguity

https://godbolt.org/z/vjY8Wc35r

```
template <typename T>
#ifdef __cpp_concepts
requires std::copy_constructible<T> && requires (T a) {
    a < a ;
    { a < a } -> std::convertible_to<bool>;
} && (!std::is_pointer_v<T>)
#endif
T min(T left, T right) {
  return left < right ? left : right;
}
template<typename T, size_t N, size_t M>
auto min(T const (&left)[N], T const (&right)[M]){
    return std::lexicographical_compare(std::begin(left),std::end(left),std::begin(right),std::end(right))?
    left:right;
}
```

© 2022 Peter Sommerlad

Overload Resolution *Magic*

When overloading functions and function templates the best fitting match wins

ambiguity errors occur, when two are equivalent best

0. name lookup for the function name, including ADL -> set of candidate function(-template)s, if more than one candiate, then:
1. exact match of non-template function
2. match of non-template function after decay or implicit conversion
3. match of function template without need for conversion

the above is a simplification!

© 2022 Peter Sommerlad

Speaker notes

Overload resolution is magic, meaning it might select something non-obvious, or it might not work due to ambiguities. OTOH, when it works it shines and allows us to write generic code in templates or generic lambdas.

As general rule: the best match wins.

Named lambda objects cannot be overloaded, so they do not suffer the potential problems of overloading but also lack its benefits.

C++20 concepts increase the abilities to disambiguate overload selection of function templates. Similar effects can be achieved in earlier versions with more complicated means (`std::enamble_if`), we don't cover here.

© 2022 Peter Sommerlad

Arbitrary Number of Parameters

```
printAll(std::cout,1);
printAll(std::cout, "the answer is ", 6*7);
auto const number = 2;
printAll(std::cout, 5, " times ", number, " is ", 5 * number, '\n');
```

How can we implement such a function in a type-safe way?

could try to create overloads for 1, 2, 3, 4, 5, 6, ..? parameters

```
template<typename First>
void printAll(std::ostream & out, First const & first);
template< typename First, typename Second>
void printAll(std::ostream & out, First const & first, Second const & second);
//... how many?
template<typename First, typename Second, typename Third,
     typename Fourth, typename Fifth, typename Sith>
void printAll(std::ostream & out,
    First const & first, Second const & second,
    Third const & third, Fourth const & fourth,
    Fifth const & fifth, Sith const & sith);
```


© 2022 Peter Sommerlad

Speaker notes

Before C++11 it was necessary to work that way and provide that many overloads up to an upper limit. This caused large overload sets, and potentially slower compile times in large systems.

© 2022 Peter Sommerlad

Variadic function templates

```
void printAll(std::ostream & out) { // recursion base case
  out << '\n';
}
template<typename First, typename...Types>
void printAll(std::ostream & out, First const & first, Types const &...rest) {
  out << first;
  if (sizeof...(Types)) {
    out << ", ";
  }
  printAll(out, rest...);
}
```

- `typename...Types` - define template parameter pack
- `Types const &...rest` - define function parameter pack
- `rest...` - expand parameter pack (comma separated)
- `sizeof...` - determine number of elements in pack

© 2022 Peter Sommerlad

Speaker notes

Before C++17, the only way to implement variadic function templates well, was recursion.

The code above misses the case, when recursion leads to an empty parameter pack, then the minimum of 2 parameter that are required no longer match.

Therefore, we need a recursion base case to be defined as well and before the variadic function template:

```
void printAll(std::ostream & out) { // recursion base case
  out << '\n';
```

Failed attempt in trying to prevent recursion:

https://godbolt.org/z/MvGEsbnfa

`if constexpr` helps:

https://godbolt.org/z/svc6qxeh8

observe template instantiations:

https://cppinsights.io/s/7d422e0f

© 2022 Peter Sommerlad

Failed variadic recursion

```
template<typename First, typename...Types>
void printAll(std::ostream & out, First const & first, Types const &...rest) {
  out << first;
  if (sizeof...(Types)) {
    out << ", ";
    printAll(out, rest...);
  } else {
    out << '\n';
  }
}
```

- even so, the function would not be called when the parameter pack is empty, it must compile

```
/Users/sop/Documents/talks/Firmenkurse/workshop/sources/CPPCourseAdvan
 error: no matching function for call to 'printAll'
        printAll(out, rest...);
```


© 2022 Peter Sommerlad

Speaker notes

Failed attempt in trying to prevent recursion:

https://godbolt.org/z/MvGEsbnfa

© 2022 Peter Sommerlad

Failed recursion - corrected

```
template<typename First, typename...Types>
void printAll(std::ostream & out, First const & first, Types const &...rest) {
  out << first;
  if constexpr (sizeof...(Types)) { // compile-time if
    out << ", ";
    printAll(out, rest...);
  } else {
    out << '\n';
  }
}
```

`if constexpr` *is a compile-time conditional statement*

https://godbolt.org/z/svc6qxeh8

© 2022 Peter Sommerlad

Speaker notes

`if constexpr` helps:

https://godbolt.org/z/svc6qxeh8

Before C++17, only the preprocessor allowed code exclusion (*`#if #ifdef #ifndef #else #endif`*). With **`if constexpr`** compile-time conditions can exclude/include parts of code during template instantiations (or for specific compile-time properties that can vary). The code in the discarded part (either if or else) must be syntactically complete, only semantic analysis will not occur, i.e., name and type resolution.

© 2022 Peter Sommerlad

Fold Expressions

```
template<typename...Types>
void printAll(std::ostream & out,  Types const &...params) {
  (out << ... << params) << '\n';
}
```

- `(init op ... op pack)` - binary left fold
- `(pack op ... op init)` - binary right fold
- `(... op pack)` - unary left fold
- `(pack op ... )` - unary right fold

fold expression must be enclosed in `( )`

https://godbolt.org/z/ra81x7vE1

© 2022 Peter Sommerlad

Speaker notes

C++17 introduces fold expressions to implement variadic function templates without the need for recursion.

Note: Cevelop/Eclipse CDT does not understand fold expression syntax yet, so don't be confused by the orange syntax-error markings.

One can even get a comma in between the elements, but that requires a bit of an ugly syntax:

```
template<typename...Types>
void printAllCommaSeparated(std::ostream & out,  Types const &...params) {
  bool first{true};
  (out << ... << ((first?(first=false):bool(out << ", ")), params)) << '\n';
}
```

This uses a flag to suppress the comma output in the first case and it uses the comma-operator to separate the right-hand fold part including the parameter pack.

https://godbolt.org/z/ra81x7vE1

observe the template instantiations with cppinsights:

https://cppinsights.io/s/ac370947

© 2022 Peter Sommerlad

Variadic Lambdas

cannot use recursion

```
auto const suml{ [](auto ... xs){ return (... + xs);} };
auto const sumr{ [](auto ... xs){ return (xs + ... );} };
```

fold expressions for adding arbitrary values

```
auto const printAll {
[](std::ostream & out,  auto const &...params) {
  (out << ... << params) << '\n';
}};
```

https://godbolt.org/z/x9eGevnWG

© 2022 Peter Sommerlad

Speaker notes

Fold expressions ease the implementation of variadic lambdas, because they do not allow recursion (until C++23 with “deducing this”)

Play with it and change the operator in suml and sumr to one that is not commudative (e.g. `/`) and observe the difference:

https://godbolt.org/z/x9eGevnWG

Use cppinsights to see how fold expressions are expanded:

https://cppinsights.io/s/94efa5a9

© 2022 Peter Sommerlad

Generic Functions summary

- Function templates allow to write code usable with all types that fulfil its concept
- For handling special cases specific overloads can be required
- C++20 allows to explicitly state template parameter's concepts
  - previous versions check concept conformance at instantiation time
- concepts allow disambiguate function template overloads
- Lambdas with **`auto`** parameters are function templates
  - cannot be overloaded
- Variadic templates and lambdas require recursion or fold expressions

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Exercise 2

exercises/exercise02.md

© 2022 Peter Sommerlad

Speaker notes

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise02.md

© 2022 Peter Sommerlad

Class Types Advanced

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

© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Simple Class Types - Refresher

- prefer `struct` over `class`
- member variables with initializer
- member functions marked with **`const`**
  - or **`&`** for side effects
- private members for strict encapsulation
- constructors for consistent init
  - resurrect default constructor with **`=default;`**
  - **`explicit`** for single argument constructors
- extending base classes, polymorphic base classes

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

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;
}
```


© 2022 Peter Sommerlad

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.

© 2022 Peter Sommerlad

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;
}
```


© 2022 Peter Sommerlad

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).

© 2022 Peter Sommerlad

A `class`' Class-Members

- member variables belong to an object of a class type
- member functions can be called on an object
- **`static`**/class variables exist per class type
- **`static`**/class member functions can be called without an object

***`static`** before a member makes it a class member*

- **`static`** data members should be **`const`**
- **`static`** member functions not subject to ADL, because of qualification

© 2022 Peter Sommerlad

Speaker notes

ADL = Argument-Dependent Lookup

A **`static`** member function can be called by qualifying it with the class name: `Classname::function()`

Normal data members are also called **non-static data member**, the inialization code is therefore also called **NSDMI** (non-static data-member initialization).

A non-const static data member is (almost) like a global variable and carries the risk of undefined behavior due to a data race in multi-threaded code!

It is sometimes better to have class related functionality as normal functions outside of the class within the same namespace.

Sometimes factory functions are implemented as static member functions, because those have access to otherwise protected functionality.

There are no **`static`** type, type alias, member class templates, member variable templates.

© 2022 Peter Sommerlad

Out-of-the box functionality

```
struct X {
  int i{};
};
X x{};       // default init
X x1{1};     // aggregate init
X x2{x1};    // copy init
X x3{X{42}}; // move init

x = x1;  // copy assignment
x = X{42}; // move assignment
```


© 2022 Peter Sommerlad

Speaker notes

If you do not define any specific constructor the compiler provides functionality to initialize or assign a value of the given type.

© 2022 Peter Sommerlad

Implicit "Special" Members

```
struct X {
  int i{};
  X() = default; // defaulted default constructor
  X(X const &x):i{x.i}{} // copy constructor
  X& operator=(X const &x){ i = x.i; } // copy assignment
  X(X&& x) noexcept : i{std::move(x.i)}{} // move constructor
  X& operator=(X &&x) noexcept { i = std::move(x.i); } // move assignment
  ~X() {} // destructor
};
```

this is a simplification

move operations are explained later, think of them as copy

© 2022 Peter Sommerlad

Speaker notes

In addition to the above 6 special member functions, an aggregate type (one with only public data members and bases and no user-declared special member functions can be initializes through aggregate initialization, by listing the initial values per subobject in `{}`.

You will not define any of these special member functions unless you must, and then only a few combinations are useful.

Move operations are optimized copy operations from temporary objects. For many types the behavior of move is the same as copy. Only those types, that manage memory, e.g., `std::vector` can benefit from move operations by transferring ownership of the managed memory instead of copying it.

The above code simplifies some things, such as omitting implicitly provided **`constexpr`** for the special member functions.

The assignment operators will have a side effect on the **`*this`** object. With that information, what can you see is wrong with the compiler-provided definitions?

© 2022 Peter Sommerlad

Copy Operations

```
struct X {
//...
  X(X const &x):i{x.i}{}                // copy constructor
  X& operator=(X const &x){ i = x.i; } // copy assignment
//...
};
```

- Parameter: *class-type* **const &**
- copies all base-class and member-variable sub-objects
- no need to define your own (except for *General Manager* classes)

Refrain from defining member variables as `const`!

© 2022 Peter Sommerlad

Speaker notes

We will introduce and look at Manager types later.

Copy assignment will not be available, if a data member or a base class prevents change. For example, if a data member is defined as `const`, then no assignment to objects of the class type can happen, ergo, **Refrain from defining member variables as `const`!**

Neither copy operation is available, if a data member or base class is preventing copying, e.g. a polymorphic base class, or if a move operation is defined in the class.

© 2022 Peter Sommerlad

Killer-feature: Destructor and }

```
struct X {
  // ...
  ~X() {} // destructor
};
```

- C++ has a deterministic object lifetime
  - temporaries die at ;
  - local variables die at }
  - global variables die at program termination
  - sequence of death is reverse to sequence of creation
- Destructors allow clean-up code to run on object death

© 2022 Peter Sommerlad

Speaker notes

- temporary objects are destroyed at the end of the *full-expression* that created them, usually at ; or in case of a condition in an **if** or **while** statement at the closing)
- local variables (automatic storage duration) are destroyed at the end of the defining block
- sequence of destruction of objects is reversed with the sequence of construction
  - member variables are destroyed in reverse order of definition
  - base class subobjects are destroyed after a class' member variables and themselves in the reverse order of definition.

© 2022 Peter Sommerlad

Privileged Constructor
a C++11 mis-design

A constructor taking a ***`std::initializer_list<T>`***
takes priority when called with ***`{values}`***

```
struct lottery_numbers {
  lottery_numbers(std::initializer_list<int> li):theNumbers{li}{}
  lottery_numbers(int first, int second):theNumbers{first,second}{}
  explicit lottery_numbers(int first):theNumbers(first){}

  lottery_numbers list1{1,2,3,4,5,6}; // braces
  lottery_numbers two{42,43}; // braces
  lottery_numbers one(0); // round ()
```

which constructor is called?

https://godbolt.org/z/hTdqn4jrP

© 2022 Peter Sommerlad

Speaker notes

When using braces `{}` to enclose a homogeneous list of values and the class to be initialized defines a constructor with a single parameter of type std::initializer_list where T is the type of the values (or one convertible to from the values), then this constructor takes priority over all other constructor overloads. To access these alternative overloads the constructor must be called with round parentheses `(values)`. A frequently used standard library class suffering from that behavior is `std::vector<int>`, where the constructors taking a size and initial values must be called with round parentheses.

The `initializer_list` special handling was introduced to get in line with aggregate initialization, where superfluous braces can be elided. However, this special casing made using classes defining such constructors with additional overloaded constructors harder to use and can cause subtle problems that you can experiment with at

https://godbolt.org/z/hTdqn4jrP

Class design rule with respect to initializer lists:

Either define a sole constructor taking a `std::initializer_list` or refrain from using `std::initializer_list`

© 2022 Peter Sommerlad

std::initializer_list-constructor Alternative

```
struct lottery_numbers {
  template<typename ...Ints>
  lottery_numbers(Ints... li)
  :theNumbers(std::initializer_list<int>{li...})// force initializer_list
        ctor
    {}
```

Define a constructor as a variadic function template

© 2022 Peter Sommerlad

Speaker notes

A compile error occurs when not all elements are homogeneous, because of the direct mapping to `std::initializer_list<int>`

© 2022 Peter Sommerlad

Move: "steal the guts"

A📪=B📪 👉 A📪, B📪

```
struct X {
  std::string s{};
//...
  X(X&& x) noexcept : s{std::move(x.s)}{} // move constructor
  X& operator=(X &&x) & noexcept {
    s = std::move(x.s);
    return *this;
  } // move assignment
};
```

- use *rvalue-reference* parameter (**&&**)
- side-effect on argument
- argument is a temporary
  - or pretends to be a temporary `std::move()`
- used for managers (`std::string`, `std::vector`)

© 2022 Peter Sommerlad

Speaker notes

While *lvalue-reference* parameters bind to variables, so you need to pass a variable(*lvalue*) as an argument, *rvalue-reference* parameters bind to temporary objects (*rvalue*) that do not have a name.

To call a function that takes an *rvalue-reference* parameter with an lvalue, you have to pass that lvalue `var` wrapped in a call to `std::move(var)`. This entitles the called function to "steal the guts" from the passed argument, because the caller implicitly promises to not use its content afterwards. The variable might be destroyed or re-assigned. Usually, such a variable is in an "empty" state, but that is not guaranteed. Internally, `std::move()` acts as a cast and does not have any overhead (except on some non-optimized codegen for debugging on some bad compilers).

Because the parameter of the move operations has a name and thus becomes an lvalue (even so the type is an rvalue-reference), one needs to use `std::move()` to pass it on, e.g., for initializing a subobject. Note, that for `int` as in the example, any move operation is a copy; for `std::string` a move operation might leave the string empty or might just be a copy. The latter happens when the *small-string optimization* SSO is applied.

Whenever we define our own assignment operators we use ref-qualification (`&`) after the function parameters to mark the side-effect on the `*this` object!

© 2022 Peter Sommerlad

What Classes We Design and How

a mental model for class design

© 2022 Peter Sommerlad

Speaker notes

Let me introduce a mental model for you to better understand what kinds of classes we design and when to define these special member functions.

© 2022 Peter Sommerlad

What C++-objects model?

Roles:

C++ specific:

© 2022 Peter Sommerlad

Speaker notes

These are the categories I'd like to talk about today. A type of an object can be simultaneously serve to more than one category

For example, providing a relation to a value object makes the latter a subject even if it holds a value, because now its location is important.

© 2022 Peter Sommerlad

What C++-objects model?

Roles:

- **Value** - what

C++ specific:

© 2022 Peter Sommerlad

What C++-objects model?

Roles:

- **Value** - what
- **Subject** - here

C++ specific:

© 2022 Peter Sommerlad

What C++-objects model?

Roles:

- **Value** - what
- **Subject** - here
- **Relation** - where

C++ specific:

© 2022 Peter Sommerlad

What C++-objects model?

Roles:

- **Value** - what
- **Subject** - here
- **Relation** - where

C++ specific:

- **Manager** - clean up

© 2022 Peter Sommerlad

What is a Value 💰?

A value is an intangible individual that exists outside time and space, and is not subject to change.
– Michael Jackson

© 2022 Peter Sommerlad

Speaker notes

I am not citing the famous US pop singer, but a Brit who was talking about IT system analysis

© 2022 Peter Sommerlad

What is a Value 💰?

A value is an intangible individual that exists outside time and space, and is not subject to change.
– Michael Jackson

When in doubt, do as the `int`s do! – Scott Meyers

© 2022 Peter Sommerlad

outside time 🕰️ and space 🌌

a value can have different representations

- `42`
- `0b10'1010`
- `052`
- `0x2A`

behavior is independent of representation and location

a value object is valid independent of other entities

© 2022 Peter Sommerlad

Speaker notes

the self-containedness of value objects is important. There is no “Fernwirkung” possible.

© 2022 Peter Sommerlad

Value Semantics in C++

property of a type

- **copyability**
  - both copy and original behave the same
  - original is unchanged by copy
- all C++ defaults support types with value semantics
- C++ built-in types have value semantics

© 2022 Peter Sommerlad

Speaker notes

value semantics does not necessarily mean instances of a type are values

For example, pointer types have value semantics, but a pointer is useless, when its target is gone

© 2022 Peter Sommerlad

What is a Subject📌?

*I choose **subject** over object*

- **identity** is important
  - has location
  - and lifetime
- in general not copyable
- target of a Relation 👉📌
- allows polymorphic behavior

object becomes a subject, once a Relation is formed to it

© 2022 Peter Sommerlad

Speaker notes

*I chose **subject** over object, because that has too many meanings*

Once we form a relation to an object, it becomes a “subject”. I chose this name, because “object” is already too overloaded and has different meanings in programming languages, e.g., C++ object means, a memory location with a type and value, in Java an object means, an instance of a class type inheriting from java.lang.Object.

Because identity is important, lifetime becomes important, because relation objects referring the subject become invalid when it is gone, or sometimes, when it is changed.

© 2022 Peter Sommerlad

Polymorphic subject types

This deals with the C++ way of using ***`virtual`***

- derived classes from a base with virtual member functions
- heap clean-up via defining virtual destructor in base
- copy-prevention via base (no value semantics)
  - keep identity 📌
  - prevent slicing 🗡️

other means for dynamic polymorphism not shown today

© 2022 Peter Sommerlad

Speaker notes

There are other means to implement dynamic polymorphism that do not rely on inheriting from a class hierarchy and that might even provide value semantics on its objects. However, those are topics for another talk and wouldn't fit within this talk's slot.

© 2022 Peter Sommerlad

What is a Relation 🔗👉?

- represents a subject 📌 (to)
  - uses its identity
- enables *"Fernwirkung"*
- enables abstraction (polymorphism)
- enables use of non-copyable objects
- most useful as parameter type

© 2022 Peter Sommerlad

Speaker notes

let me introduce a nice German word “Fernwirkung”. It can be combined with other words to become even more interesting and it enables access of the same subject from different places

It means to effect something remote/non-local from an expression.

While values act locally in the expression they are used in, using a relation object means, it can access or modify an object (its subject) that is not actually or directly part of the expression.

© 2022 Peter Sommerlad

“Fernwirkung” 🕹️

access or modify an object that is not part of current expression

- for reference parameters
  - T `const &` - access, copy-optimization
  - T `&` - side effect
  - T `&&` - transfer of ownership
- similarly for pointers, `span`, views, iterators

© 2022 Peter Sommerlad

Speaker notes

We will learn more about relation types soon.

© 2022 Peter Sommerlad

Technicalities of Relation Types 🔗

- rely on the existence of the referred entity
  - can be or become invalid: dangling or empty 💣
  - aka DANGs = potentially dangling types 🧨💥🔥
- require programmer care to track validity 🤦
  - safe to use as function parameters
- language relation types: T**&** and T*****
  - iterators, `span`, views
- Relation members make class a Relation type (contagious 🦠)
  - unless class is a Manager 👔

© 2022 Peter Sommerlad

Speaker notes

language pedants will note that reference types do not form a C++ object in a technical sense. I am aware of that, but don't want to be hair-splitting in this explanation, because other relation types, such as pointers or span actually form C++ objects with similar problems than the C++'s reference types.

© 2022 Peter Sommerlad

What is a Manager 👔?

*Manage a **single** resource*

- **Scoped Manager** 👔📦
  - Local usage of resource
- **Unique Manager** 👔🏝️
  - Resource cannot be duplicated
- **General Manager** 👔💰
  - Resource can be duplicated

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Technicalities of Managers 👔

- class defines a non-empty destructor
- usually have a member of Relation type
  - sometimes disguised, e.g., file handle `int`
- care about copying and moving

© 2022 Peter Sommerlad

Speaker notes

Think twice if you have such an odd-ball manager not actually managing a resource, caring for a non-local invariant, that might be broken by compiler-provided copy or move operations. Those often occur in bad example code demonstrating woes of move/copy operations and should rarely occur in real life. If so, they tend to try to provide “caching” of information from previous operations.

© 2022 Peter Sommerlad

Technicalities of Managers 👔

- class defines a non-empty destructor
- usually have a member of Relation type
  - sometimes disguised, e.g., file handle `int`
- care about copying and moving

never define a destructor with an empty body

© 2022 Peter Sommerlad

Kinds of Manager Types

- **Scoped Manager** 👔📦
  - Non-copyable, non-movable
  - can be returned from factory functions (>C++17)
- **Unique Manager** 👔🏝️
  - Move-only, Transfer of ownership
  - Resource can not be easily duplicated
- **General Manager** 👔💰
  - Copyable, possibly Move-operation for optimization
  - Resource can be (expensively?) duplicated

© 2022 Peter Sommerlad

Speaker notes

We will look at manager types in detail later, after we looked more deeply at relation types.

© 2022 Peter Sommerlad

Exercise 3

exercises/exercise03.md

© 2022 Peter Sommerlad

Speaker notes

TODO: tracer

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise03.md

© 2022 Peter Sommerlad

Remember What C++-objects model?

Roles:

- Value - what
- Subject - here
- **Relation** - where

C++ specific:

- Manager - clean up

© 2022 Peter Sommerlad

Speaker notes

We have seen relation types in the style of built-in references and iterators provided by standard library containers in the C++ Introduction.

Now let us look into more of it.

© 2022 Peter Sommerlad

What is a Relation 🔗👉?

- represents a subject 📌 (to)
  - uses its identity
- enables *“Fernwirkung”*
- enables abstraction (polymorphism)
- enables use of non-copyable objects
- most useful as parameter type

© 2022 Peter Sommerlad

Speaker notes

let me introduce a nice German word “Fernwirkung”. It can be combined with other words to become even more interesting and it enables access of the same subject from different places

It means to effect something remote/non-local from an expression.

While values act locally in the expression they are used in, using a relation object means, it can access or modify an object (its subject) that is not actually or directly part of the expression.

© 2022 Peter Sommerlad

“Fernwirkung” 🕹️

access or modify an object that is not part of current expression

- for reference parameters
  - T `const &` - access, copy-optimization
  - T `&` - side effect
  - T `&&` - transfer of ownership
- similarly for pointers, `span`, views, iterators

© 2022 Peter Sommerlad

Speaker notes

We will learn more about relation types soon.

© 2022 Peter Sommerlad

Technicalities of Relation Types 🔗

- rely on the existence of the referred entity
  - can be or become invalid: dangling or empty 💣
  - aka DANGs = potentially dangling types 🧨💥🔥
- require programmer care to track validity 🦸
  - safe to use as function parameters
- language relation types: `T&` and `T*`
  - iterators, `span`, views
- Relation members make class a Relation type (contagious 🦠)
  - unless class is a Manager 👔

© 2022 Peter Sommerlad

Speaker notes

language pedants will note that reference types do not form a C++ object in a technical sense. I am aware of that, but don't want to be hair-splitting in this explanation, because other relation types, such as pointers or span actually form C++ objects with similar problems than the C++'s reference types.

© 2022 Peter Sommerlad

C++ built-in relation types

Type	Name	Usage
T `&`	lvalue-reference	parameter with side-effect
T `const &`	const-reference	value parameter substitute, copy optimization
T `&&`	rvalue-reference	transfer-of-ownership parameter for gut stealing
T `*`	raw pointer	optional reference to mutable single object, (optional) C-style array of T with indeterminate bound (its begin() iterator)
T`[]`	raw pointer	array of indeterminate bound different syntax for T`*`
T `const *`	raw pointer	optional reference to immutable single object (optional) C-style array of immutable T with indeterminate bound (its cbegin() iterator)
`char const *` `char *`	C-style string	by convention beginning of C-style string with NUL '\0' termination, beginning of raw buffer (writable w/o const)

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Raw Pointers 🧨🎇🔥

```
T * rawpointer;
```

- Interface with C code
- Legacy usage (see later for replacement options)
- Allow too many operations - iterator protocol
- can mean too many things (optional single object, array)
- replace or encapsulate safely
- special convention for pointer to `char`
- **`nullptr`** value means absence

© 2022 Peter Sommerlad

Speaker notes

While C code and old C++ code can be dominated by the use of pointer types, good C++ makes their use rare and tightly encapsulated.

Within library code or when needing direct access to hardware, raw pointer types can be required, but should be tightly encapsulated.

The C++ Core guidelines assume that a raw pointer value means “an optional single object reference”, but that convention is not helpful when maintaining or renovating existing code that does not follow this. The C++ library fundamentals technical specification v2 tried to introduce a library type for that convention case unter the name `std::experimental::observer_ptr<T>` but unfortunately, there was no consensus to migrate that into the official C++ standard (https://en.cppreference.com/w/cpp/experimental/observer_ptr)

© 2022 Peter Sommerlad

Random Access Iterator and T*

random access iterators were modeled after pointers

- `*it` - element access
- `++it it++ --it it--` - moving around
- `it[5]` - indexed element access
- `it += 5; it -= 3; *(it+3)` - jump around
- `it == it2 || it != it1` - equality comparison
- `it < it2 || it > it1` - relational operators (only valid if same range)
- `it2 - it1` - `distance(it1, it2)`
- `std::iterator_traits<T*>::iterator_category` is `std::random_access_iterator_tag`

too many operations supported

© 2022 Peter Sommerlad

Speaker notes

Pointers support all operations of a random access iterator. Without being extremely careful, these operations can lead to accessing memory outside of the underlying array object and thus cause undefined behavior in addition to the problem of dangling.

A pointer variable can happen to be uninitialized, thus containing an invalid value. When used -> undefined behavior.

A pointer variable can be initialized or assigned to the **`nullptr`** value. This allows to mark the absence of a pointed-to object. hence the usability as an optional reference as suggested by the C++ core guidelines. like `std::optional`, such a pointer must be checked for validity, before the referred object is accessed.

Only a few algorithms actually require a random access iterator, and those are related with sorting:

- shuffle()
- sample() (under specific conditions)
- sorting: sort(), partial_sort(), partial_sort_copy(), stable_sort(), nth_element()
- heap: is_heap(), is_heap_until(), make_heap(), push_heap(), pop_heap(), sort_heap()

However, some algorithms can benefit from being passed random access iterators for improved performance or the possibility of vectorization (if memory is contiguous) or parallelization.

© 2022 Peter Sommerlad

C-style Arrays

Do not use C-style arrays

```
int a[10]={1,2,3,4,5}; // 10 ints with non-zero initial values for the
        first 5
func(a); // passes int *, the address of the first element &a[0]
```

when used as function argument, dimension is lost

- need to pass number of elements separately
- or have other convention, like `char *` with `'\0'`
- indexing from zero, never checked

© 2022 Peter Sommerlad

Speaker notes

Many programming tutorials begin with the data structure of an array. Syntactical plain arrays in C++ are of no good use, because they have a multitude of usability problems.

The concept of a *variable-length array* (VLA) of C is unavailable in standard C++ (some compilers provide it as an extension), because it does not fit into C++'s rigorous compile-time checked type system.

To correctly pass a plain int array by value one needs to declare func as follows

```
template<size_t N>
void func(int (&a)[N]){...}
// or for immutable elements:
template<size_t N>
void func(int const (&a)[N]){...}
```

Pass the array by lvalue-reference and deduce the dimension. If the latter is fixed, one can also specify the dimension directly but still must pass the array by reference and use parentheses, because the array brackets would bind before the reference and one cannot declare an array of references.

© 2022 Peter Sommerlad

`std::string_view`

C++ replacement for `char const *` parameters

- C-style strings use a `char *` and `'\0'` as implicit end sentinel
- `"Hello"` used as argument becomes a `char const *`
- implicit length needs to be recovered by scanning memory
- `std::string_view` explicitly provides length
- prefer `string_view` over char const * for C++ code
  - unsuitable when need to call C-APIs, no `'\0'`
- when interfacing to C code prefer `std::string`
  - `std::string::c_str()` provides a C-style `char const *`

© 2022 Peter Sommerlad

Speaker notes

std::string_view was introduced as an alternative to std::string as parameter type, because it is efficient to copy and provides the length of the string directly in contrast to a `char const *`. However, because it might refer to a substring, string_view does not guarantee that the underlying character sequence conforms to a C-style string with `'\0'` terminating character.

When a function needs to take a string and pass it on to a C-API expecting a *NUL-terminated-byte-string* (NTBS) better use std::string as the function parameter, because its member function `.c_str()` guarantees to return a NTBS compatible `char const *`.

This situation causes the standard library to provide one, two or three overloads of functions taking

- `std::string const &`,
- `std::string_view`, or
- `char const *`

as their parameter type. For your own code, I recommend that you stick with std::string and only chose `std::string_view` instead, if you do not pass the string to a C-API and the strings passed can be very very long.

© 2022 Peter Sommerlad

Arguments to main()

```
int main(int argc, char *argv[])
```

- `main()` can take arguments from the command line
- conceptually this is an array of strings
- technically `main` obtains a counter and a decayed array of `char*` representing those strings
- whenever using the arguments passed to main, I recommend that one copies them into a `std::vector<std::string>` to avoid further problems.
- argv[0] is the program name
- argv[1] is the first argument (argc>1)

© 2022 Peter Sommerlad

Speaker notes

while ideas exist to allow alternative versions, i.e., with a std::vector the standard does not define it that way.

© 2022 Peter Sommerlad

std::span<T>

C++20 `std::span` is a pointer+size parameter

- can convert from `std::array`, C-style array, or `std::vector`
- contiguous memory of `.size()` elements of type `T`
- useful for interfacing with C APIs
- single parameter instead of begin-end or ptr-size pairs
- elements can be changed, unless `span<T const>`
- similar problems as raw pointer (empty, no bounds check, dangling)
- provides raw-bytes buffer access

© 2022 Peter Sommerlad

Speaker notes

caveats of span:

- a very low-level interface type to replace pointer+size APIs
- no checked access like `.at(index)` of vector or array
- `cbegin(sp)` might provide iterator to mutable elements
- no guarantee that there are elements there
- can be dangling, especially when created from temporary
- writing into the memory area provided by `.as_writable_bytes()` does not guarantee that the underlying objects are valid

© 2022 Peter Sommerlad

Why are raw pointers dangerous?

- pointer may be indeterminate, invalidated, or dangling
- pointer may have the value `nullptr`
- pointer can change (unless `T * const pointer;`)
- pointer change can cause out-of-bounds access
- pointers and C-style arrays are a security hazard
- a pointer type can mean too many things
- “adress of memory” not guaranteed by C++ type system

© 2022 Peter Sommerlad

Speaker notes

I am so sad, that so many introductions to C++ still start out with a naked pointer(=raw pointer) world view.

Pointers are required in the language for interfacing with C APIs and hardware, but should not be used on other interfaces or within other types, unless safely encapsulated.

Pointers get invalidated when the referred object (a subject) is destroyed, even when directly a new object is created at the same memory location. Using the original pointer to access the new object is undefined behavior and might not work as expected (known as *pointer zap* and might change in the future). This is meant by “address of memory” not guaranteed. The latter only works for `std::byte *` or `char *` respectively.

© 2022 Peter Sommerlad

Pointer replacement overview

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable	value	`optional<T>`	to denote missing value best for return values
	🤷			

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable	value	`optional<T>`	to denote missing value best for return values
	🤷	heap	`unique_ptr<T> const`	T can be base class with `make_uniqe<Derived>`

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable 🤷	value	`optional<T>`	to denote missing value best for return values
		heap	`unique_ptr<T> const`	T can be base class with `make_uniqe<Derived>`
referring T 🤷	non-null	fixed	T &	can dangle

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable 🤷	value	`optional<T>`	to denote missing value best for return values
		heap	`unique_ptr<T> const`	T can be base class with `make_uniqe<Derived>`
referring T 🤷	non-null	fixed	T &	can dangle
		rebind	`reference_wrapper<T>`	assignability with a reference member

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable 👷	value	`optional<T>`	to denote missing value best for return values
		heap	`unique_ptr<T> const`	T can be base class with `make_uniqe<Derived>`
referring T 👷	non-null	fixed	`T &`	can dangle
		rebind	`reference_wrapper<T>`	assignability with a reference member
	nullable 👷	fixed	`jss::object_ptr<T> const` `boost::optional<T&> const`	missing in std `std::optional` can not do this `boost::optional` can `object_ptr<T>` by A. Williams

© 2022 Peter Sommerlad

Pointer replacement overview

owning T	non-null	value	T	most safe and useful
		heap	`unique_ptr<T> const`	must be init with `make_unique<T>`
	nullable 👷	value	`optional<T>`	to denote missing value best for return values
		heap	`unique_ptr<T> const`	T can be base class with `make_uniqe<Derived>`
referring T 👷	non-null	fixed	`T &`	can dangle
		rebind	`reference_wrapper<T>`	assignability with a reference member
	nullable 👷	fixed	`jss::object_ptr<T> const` `boost::optional<T&> const`	missing in std `std::optional` can not do this `boost::optional` can `object_ptr<T>` by A. Williams

		rebind	`jss::object_ptr<T>`
			`boost::optional<T&>`
			`optional<reference_wrapper<T>>`	

109

© 2022 Peter Sommerlad

Examples replacing plain pointers

Old/Unsafe	Modern/Better	Alternative

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Examples replacing plain pointers

	Old/Unsafe	Modern/Better	Alternative
Heap	`void demoObjectOnHeap(int arg){ Object *o = new Object{arg}; o->dosomething(); delete o; }`	`void demoObjectNotOnHeap(int arg){ Object o{arg}; o.dosomething(); }`	`void demoObjectOnHeapSafe(int arg){ auto o = std::make_unique<Object>(arg); o->dosomething(); }`

© 2022 Peter Sommerlad

Examples replacing plain pointers

	Old/Unsafe	Modern/Better	Alternative
Heap	`void demoObjectOnHeap(int arg){ Object *o = new Object{arg}; o->dosomething(); delete o; }`	`void demoObjectNotOnHeap(int arg){ Object o{arg}; o.dosomething(); }`	`void demoObjectOnHeapSafe(int arg){ auto o = std::make_unique<Object>(arg); o->dosomething(); }`
Failure Return	`Object * computeOrNullOnFail(){ return nullptr; }`	`std::optional<Object> computeThatMightFail(){ return std::nullopt; }`	`Object computeThatMightFailRef(){ throw computation_failed{}; }`

© 2022 Peter Sommerlad

Examples replacing plain pointers

	Old/Unsafe	Modern/Better	Alternative

Heap	`void demoObjectOnHeap(int arg){`
	`  Object *o = new Object{arg};`
	`  o->dosomething();`
	`  delete o;`
	`}`
		`void demoObjectNotOnHeap(int arg){`
		`  Object o{arg};`
		`  o.dosomething();`
		`}`
			`void demoObjectOnHeapSafe(int arg){`
			`  auto o = std::make_unique<Object>(arg);`
			`  o->dosomething();`
			`}`

Failure Return	`Object * computeOrNullOnFail(){`
	`  return nullptr;`
	`}`
		`std::optional<Object> computeThatMightFail(){`
		`  return std::nullopt;`
		`}`
			`Object computeThatMightFailRef(){`
			`  throw computation_failed{};`
			`}`

Single Object Parameter	`void computeWithSingleObject (Object * const o){`
	`  o->dosomething();`
	`}`
		`void computeWithObject (Object &o) {`
		`  o.dosomething();`
		`}`
			`void computeWithObjectValue (Object o) {`
			`  o.dosomething();`
			`}`

© 2022 Peter Sommerlad

Examples replacing plain pointers

	Old/Unsafe	Modern/Better	Alternative

Heap	`void demoObjectOnHeap(int arg){`
	`  Object *o = new Object{arg};`
	`  o->dosomething();`
	`  delete o;`
	`}`
		`void demoObjectNotOnHeap(int arg){`
		`  Object o{arg};`
		`  o.dosomething();`
		`}`
			`void demoObjectOnHeapSafe(int arg){`
			`  auto o = std::make_unique<Object>(arg);`
			`  o->dosomething();`
			`}`

Failure Return	`Object * computeOrNullOnFail(){`
	`  return nullptr;`
	`}`
		`std::optional<Object> computeThatMightFail(){`
		`  return std::nullopt;`
		`}`
			`Object computeThatMightFailRef(){`
			`  throw computation_failed{};`
			`}`

Single Object Parameter	`void computeWithSingleObject (Object * const o){`
	`  o->dosomething();`
	`}`
		`void computeWithObject (Object &o) {`
		`  o.dosomething();`
		`}`
			`void computeWithObjectValue (Object o) {`
			`  o.dosomething();`
			`}`

Ownership Transfer	`Object * makeObject(){`
	`  Object *o=new Object{42};`
	`  o->dosomething();`
	`  return o;`
	`}`
		`Object initObject(){`
		`  Object o{42};`
		`  o.dosomething();`
		`  return o; // NRVO`
		`}`
			`std::unique_ptr<Object> makeNewObject(){`
			`  auto o = std::make_unique<Object>(42);`
			`  o->dosomething();`
			`  return o;`
			`}`

© 2022 Peter Sommerlad

`object_ptr` by Anthony Williams

- `std::experimental::observer_ptr`
  - with a better name
- intended use:
  - nullable referencing parameter type
  - where `optional<T&>` would be nice to have
- all smart pointers automatically convert to object_ptr
- NO pointer arithmetic possible

```cpp
#include "object_ptr.hpp"
#include <iostream>

void foo(jss::object_ptr<int> p) {
    if(p) {
        std::cout << *p;
    } else {
        std::cout << "(null)";
    }
    std::cout << "\n";
}
int main() {
    foo(nullptr);
    int x= 42;
    foo(&x);
    auto sp= std::make_shared<int>(123);
    foo(sp);
    auto up= std::make_unique<int>(456);
    foo(up);
}
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Breaking Encapsulation 🎁→🗃️

💣 `smart_p.get()`

ignorance or truth

© 2022 Peter Sommerlad

Speaker notes

all smart pointers allow access to the underlying raw pointer by using the member function `get() const`. This is highly dangerous and error prone, because all limits are off. similarly `unique_ptr::release()` on a non-nullptr pointer is dangerous as well. Try to use tooling that flags those uses as errors.

© 2022 Peter Sommerlad

Pointers as Array replacement

Old/Unsafe	Modern/Better	Alternative

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Pointers as Array replacement

	Old/Unsafe	Modern/Better	Alternative

with explicit bound	`void absarray(int a[], size_t len)`
	`{`
	`  for(size_t i=0; i < len; ++i)`
	`  a[i] = a[i] < 0? -a[i]:a[i];`
	`}`
		`template<size_t N>`
		`void absarray(int (&a)[N]) {`
		`  for(size_t i=0; i < N; ++i)`
		`  a[i] = a[i] < 0? -a[i]:a[i];`
		`}`
			`// C++ 20 or GSL`
			`void absarrayspan(std::span<int> a){`
			`  for(size_t i=0; i < a.size(); ++i)`
			`  a[i] = a[i] < 0? -a[i] : a[i];`
			`}`

© 2022 Peter Sommerlad

Pointers as Array replacement

	Old/Unsafe	Modern/Better	Alternative

with explicit bound	`void absarray(int a[], size_t len)`
	`{`
	`  for(size_t i=0; i < len; ++i)`
	`  a[i] = a[i] < 0? -a[i]:a[i];`
	`}`
		`template<size_t N>`
		`void absarray(int (&a)[N]) {`
		`  for(size_t i=0; i < N; ++i)`
		`  a[i] = a[i] < 0? -a[i]:a[i];`
		`}`
			`// C++ 20 or GSL`
			`void absarrayspan(std::span<int> a){`
			`  for(size_t i=0; i < a.size(); ++i)`
			`  a[i] = a[i] < 0? -a[i] : a[i];`
			`}`

implicit sentinel nul/nullptr	`void takecharptr(char *s){`
	`  for (; *s; ++s)`
	`  *s = std::toupper(*s);`
	`}`
		`void takestring(std::string &s){`
		`  transform(s.begin(),s.end(),`
		`  s.begin(),`
		`  [](char c){`
		`  return std::toupper(c);});`
		`}`
			`void take(std::string_view s){`
			`  // can not change`
			`}`

© 2022 Peter Sommerlad

Pointers as Array replacement

	Old/Unsafe	Modern/Better	Alternative

with explicit bound	`void absarray(int a[], size_t len)`
	`{`
	`  for(size_t i=0; i < len; ++i)`
	`  a[i] = a[i] < 0? -a[i]:a[i];`
	`}`
		`template<size_t N>`
		`void absarray(int (&a)[N]) {`
		`  for(size_t i=0; i < N; ++i)`
		`  a[i] = a[i] < 0? -a[i]:a[i];`
		`}`
			`// C++ 20 or GSL`
			`void absarrayspan(std::span<int> a){`
			`  for(size_t i=0; i < a.size(); ++i)`
			`  a[i] = a[i] < 0? -a[i] : a[i];`
			`}`

implicit sentinel nul/nullptr	`void takecharptr(char *s){`
	`  for (; *s; ++s)`
	`  *s = std::toupper(*s);`
	`}`
		`void takestring(std::string &s){`
		`  transform(s.begin(),s.end(),`
		`  s.begin(),`
		`  [](char c){`
		`  return std::toupper(c);});`
		`}`
			`void take(std::string_view s){`
			`  // can not change`
			`}`

explicit range	`void absintptrrange(int *b, int *e){`
	`  for (;b != e ; ++b){`
	`  if(*b < 0) *b = - *b;`
	`  }`
	`}`
		`void absintptrrange(int *b, int *e)`
		`{`
		`  std::transform(b,e,b,`
		`  [](auto i){ return std::abs(i);});`
		`}`
			`template <typename FWDITER>`
			`void absintarray(FWDITER b, FWDITER e)`
			`{`
			`  std::transform(b,e,b,`
			`  [](auto i){ return std::abs(i);});`
			`}`

© 2022 Peter Sommerlad

Alternatives to plain arrays

`std::vector<T>`

`std::array<T,N>`

`std::string`

`string_view` 💣

`span<T>` 💣

© 2022 Peter Sommerlad

Speaker notes

`std::string_view` and `std::span` are ***relation types***, so they might dangle. Those types can only safely be used as parameter types to pass a range in a lightweight way (no copying of the range). However, returning them from a function or having a local variable of these types is almost always an error, because it is too easy to make the mistake to use the range view, when the underlying container is already gone. Even if it works today, a slight refactoring of the code can break it and cause undefined behavior.

© 2022 Peter Sommerlad

How to use Relation Types

References, Spans, Views, unmanaged Pointers

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Using references/pointers/views safely

*aka Safe Use of **Relation Types** aka **DANGs***

Being a superhero without hanging yourself

Relation Type = DANG = potentially dangling object's type

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Plain pointers have issues!

One does not know what they mean:

- single object
- `'\0'`-terminated character string
- optional object - **`nullptr`**
- out-parameter
- generic parameter `void*`
- array of objects
- memory buffer
- decayed function argument
- decayed array
- allocated memory ownership
  - to be released later
    - single object **`delete`** p
    - array **`delete[]`**

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Potentially Dangling Objects

*Validity of a **potentially dangling object** depends on lifetime of other object. A potentially dangling object has a potentially dangling type -* ***DANG***

- plain pointers (`T*`)
- references (`T&`, `T const &`, `T&&`)
- iterators (invalidation on container change)
- views (`std::string_view` `std::span` `ranges::borrowed_range`)
- implicit app-specific, i.e., index into vector

Many of following slides on pointers also are valid for DANGs

The term DANG is inspired by Tony van Eerd because it is close to BANG 💣

© 2022 Peter Sommerlad

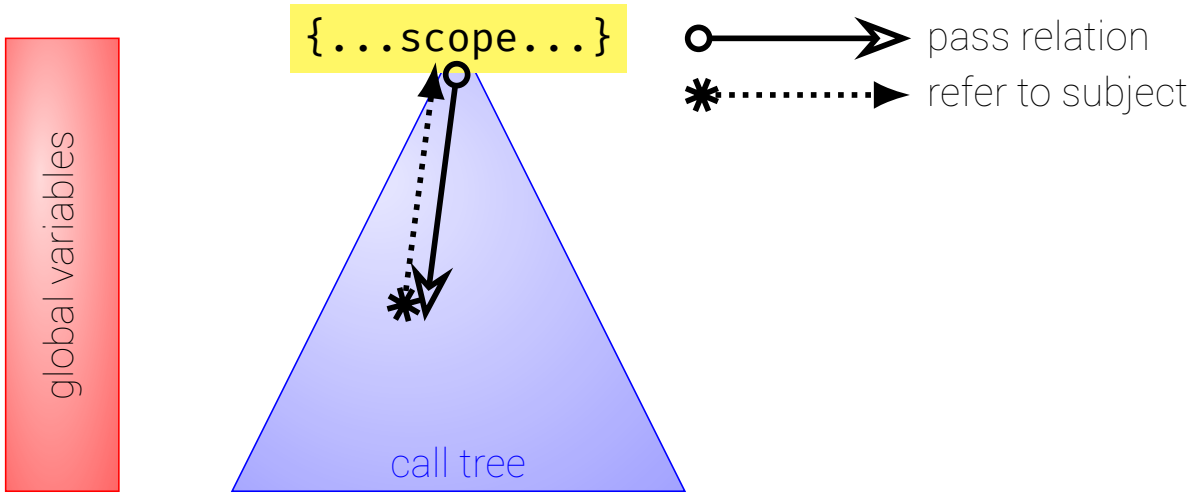
Speaker notes

for your own notes

© 2022 Peter Sommerlad

Passing DANGs

Passing references/pointers/views down the call tree is dangle free



*Relation types (e.g. `std::span`) are also known as **parameter types***

© 2022 Peter Sommerlad

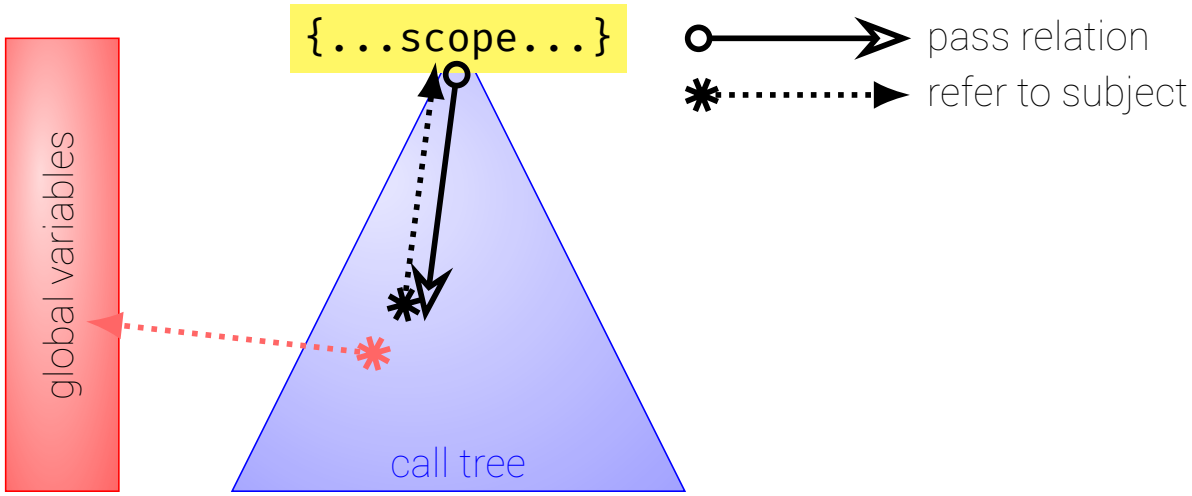
Speaker notes

using global variables is poisonous! They taint your code and make it untestable.

© 2022 Peter Sommerlad

Globals cannot dangle, **BUT…**

Better pass parameters!



Global variables *make the code untestable in isolation*

© 2022 Peter Sommerlad

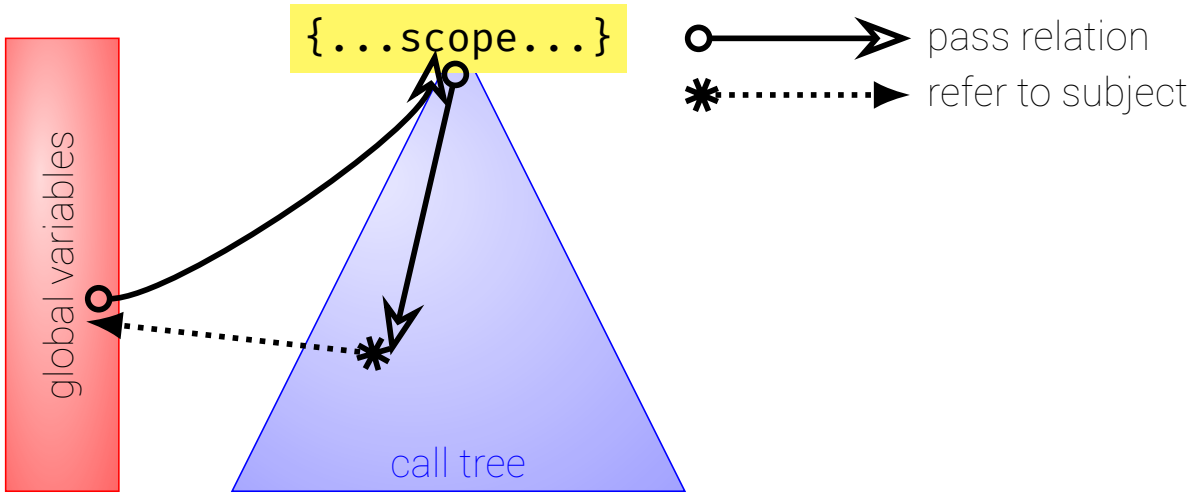
Speaker notes

using global variables is poisonous! They taint your code and make it untestable.

© 2022 Peter Sommerlad

Parameterize from Above

pass global variables as parameters from `main()`!



Enable testability by using different arguments for tests instead

© 2022 Peter Sommerlad

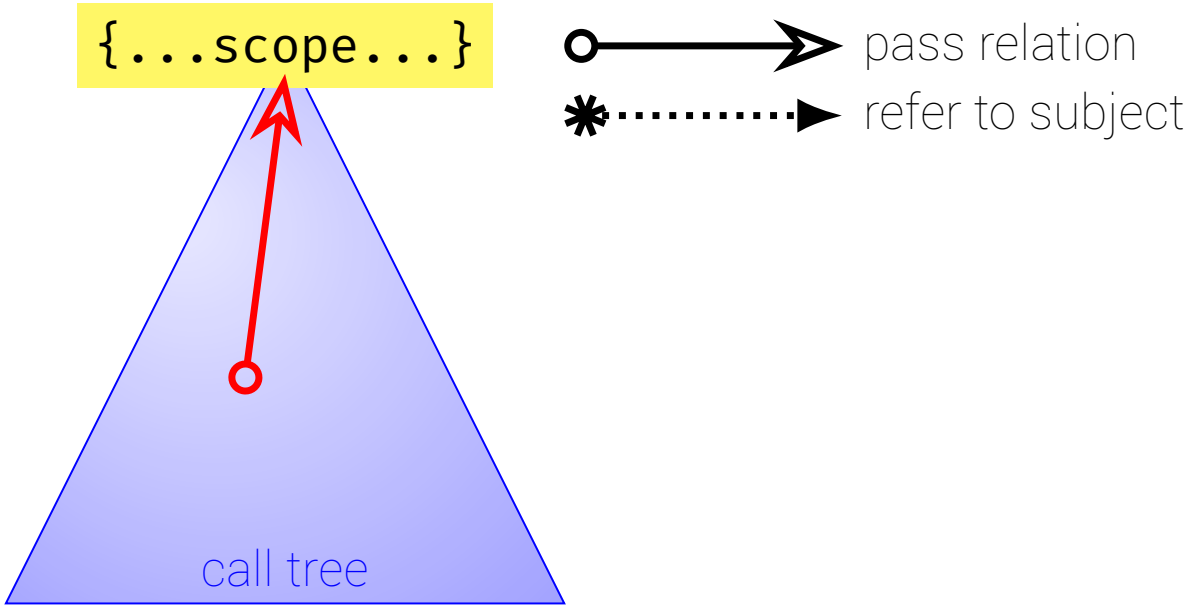
Speaker notes

using global variables is poisonous! They taint your code and make it untestable. Pass all dependencies as parameters!

© 2022 Peter Sommerlad

Don't Return Relations 🧨🎇🔥

Returning relation objects (DANGs) up the call tree can/will dangle!



© 2022 Peter Sommerlad

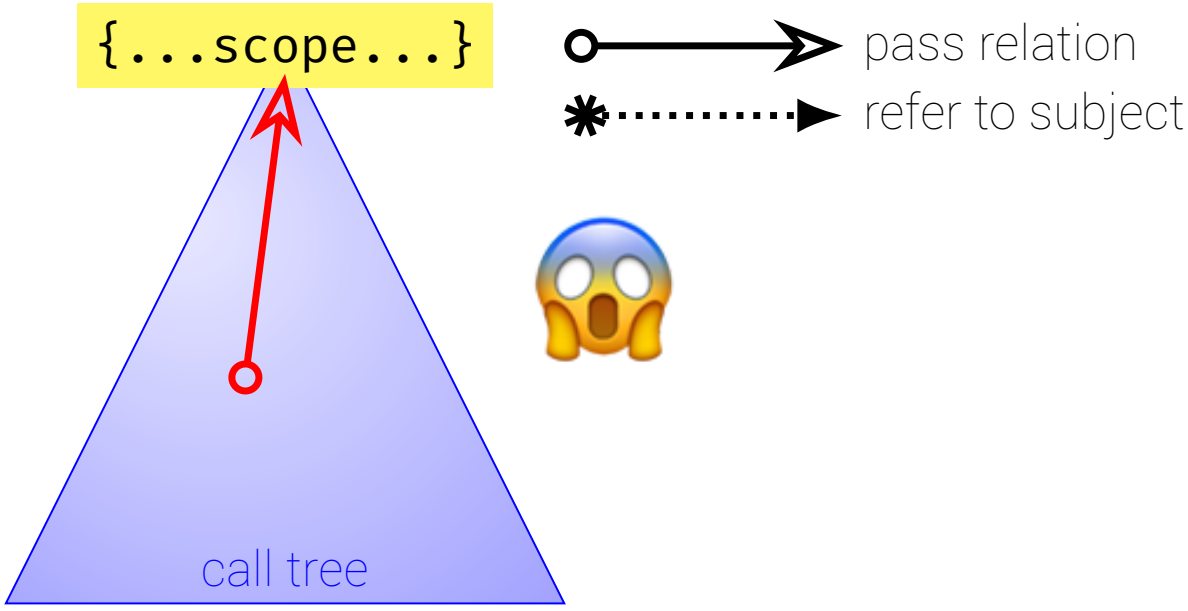
Speaker notes

returning relation objects referring local subjects will lead to immediate dangling

© 2022 Peter Sommerlad

Don't Return Relations 🧨🎇🔥

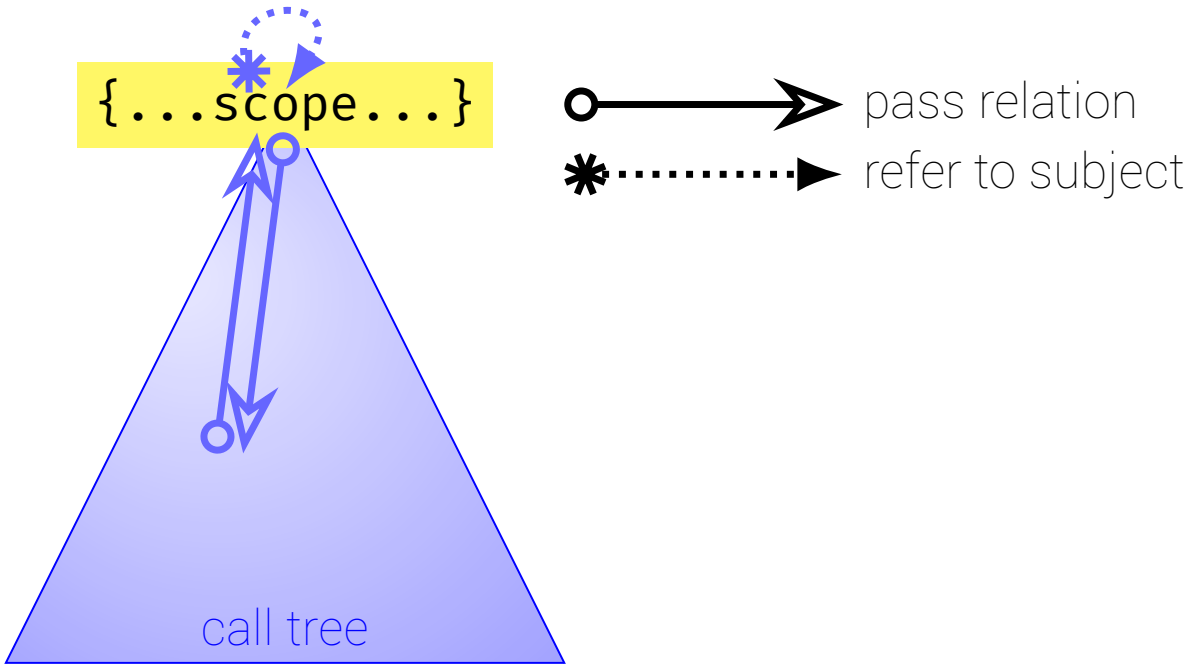
Returning relation objects (DANGs) up the call tree can/will dangle!



© 2022 Peter Sommerlad

Return only Relations to Parameters

still requires care about lifetime, subject might be a temporary!



© 2022 Peter Sommerlad

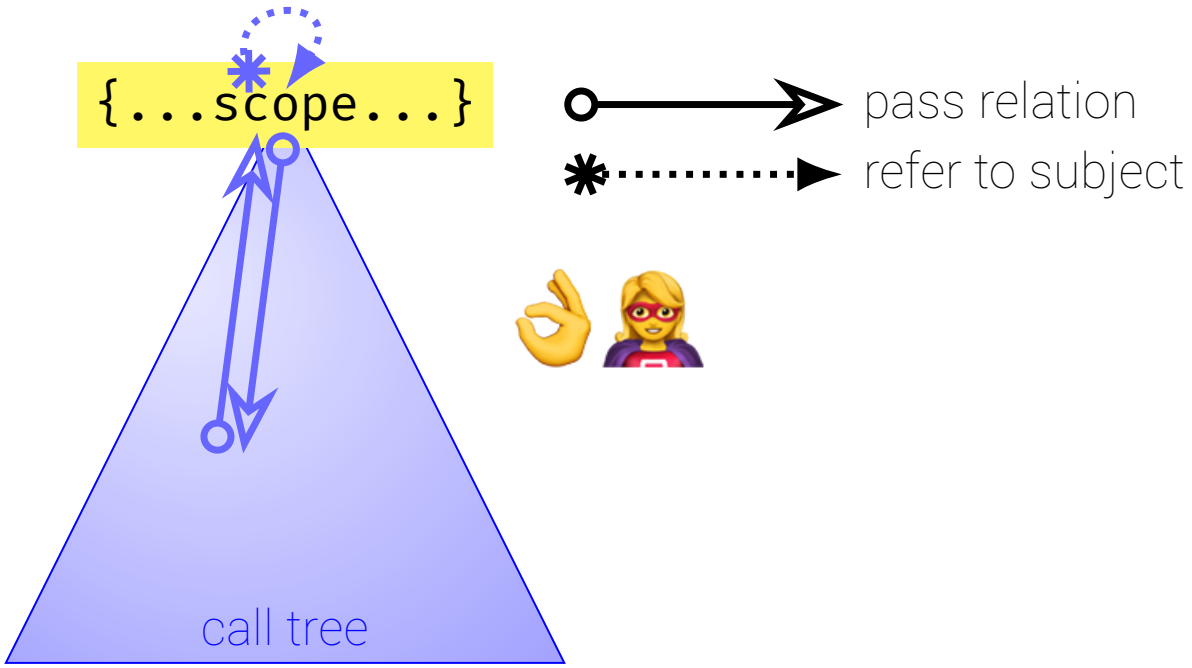
Speaker notes

returning relation objects referring local subjects will lead to immediate dangling

© 2022 Peter Sommerlad

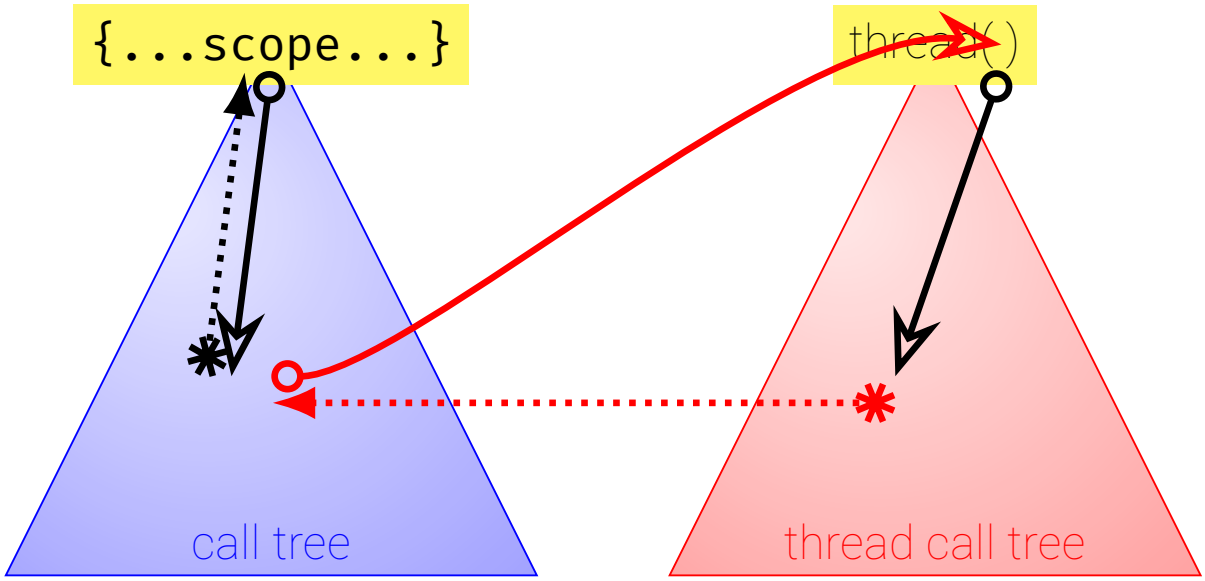
Return only Relations to Parameters

still requires care about lifetime, subject might be a temporary!



© 2022 Peter Sommerlad

Relations and Threads



*Passing Relations to another thread risks **data races***

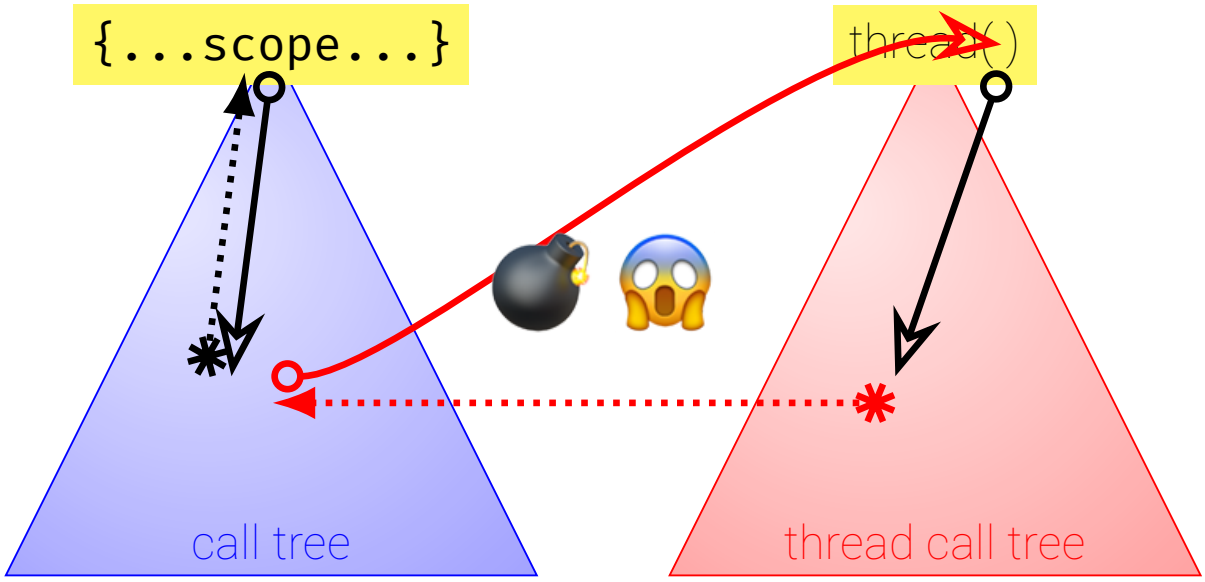
© 2022 Peter Sommerlad

Speaker notes

Pass data to thread by value. This means each thread gets its own copy and does not need to access shared data, which leads to data races and thus **undefined behavior**.

© 2022 Peter Sommerlad

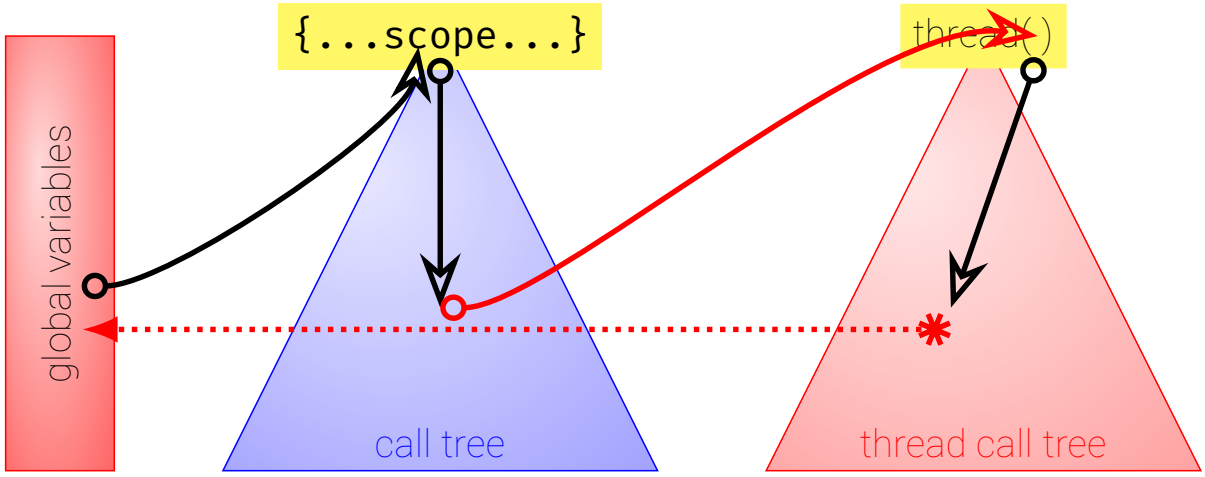
Relations and Threads



*Passing Relations to another thread risks **data races***

© 2022 Peter Sommerlad

Globals and Threads



*Using mutable global variables in multiple threads risks **data races***

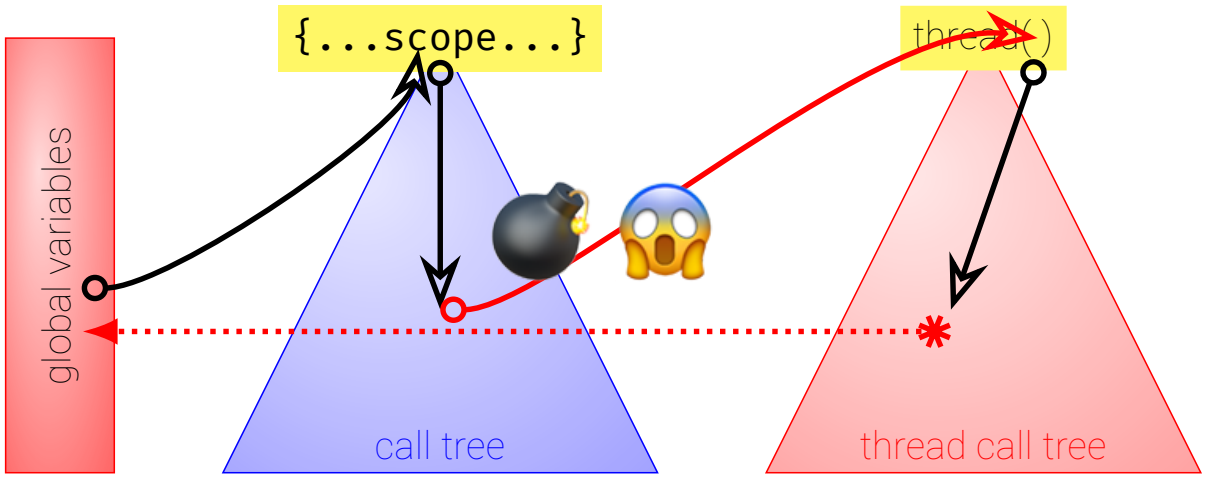
© 2022 Peter Sommerlad

Speaker notes

Mutable (non-const) global variables (with static storage duration) accessed from multiple threads cause data races and thus **undefined behavior**.

© 2022 Peter Sommerlad

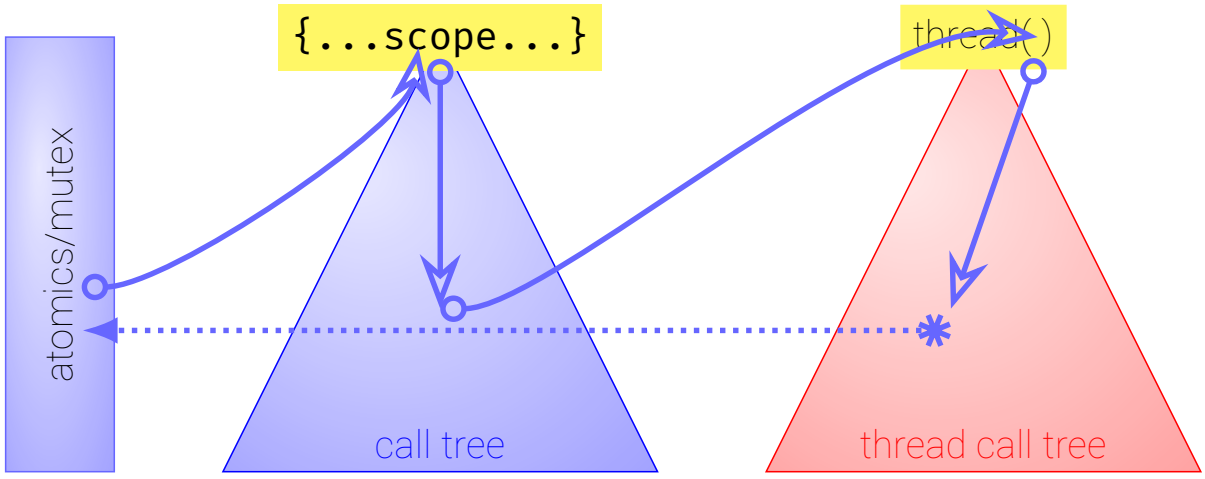
Globals and Threads



*Using mutable global variables in multiple threads risks **data races***

© 2022 Peter Sommerlad

Safer Sharing with Threads



Using atomic variables or objects protected with a mutex is required

© 2022 Peter Sommerlad

Speaker notes

Mutable (non-const) global variables (with static storage duration) accessed from multiple threads cause data races and thus **undefined behavior**.

© 2022 Peter Sommerlad

Exercise 4

exercises/exercise04.md

© 2022 Peter Sommerlad

Speaker notes

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise04.md

© 2022 Peter Sommerlad

Class Templates

`name< params >`

⟹

`value | function |`**`type`**

we are looking at the type aspect now

© 2022 Peter Sommerlad

Speaker notes

C++ templates allow abstracting functions, values (as variable templates), and types. The latter as class templates.

© 2022 Peter Sommerlad

Reminder: Template Syntax

```
template<Template-Parameter-List>
Function-Definition
```

- Templates start with keyword **`template`**
- Template parameter:
  - **`typename`**: placeholder for type
  - **`template`**: placeholder for a class template
  - **`auto`**/type: placeholder for a value

© 2022 Peter Sommerlad

Speaker notes

For legacy reasons, instead of **typename** for type template parameters the keyword **class** can also be used.

© 2022 Peter Sommerlad

Reminder: Template Instantiation

```
template <typename T>
T min(T a, T b){
        return (a < b)? a : b ;
}
```

template definition

`min(1,2)` -> TAD

`min<int>()` -> template id

```
int min<int>(int left, int right) {
  return left < right ? left : right;
}
```

template instance

© 2022 Peter Sommerlad

Speaker notes

TAD: template argument deduction

`min<int>` is called the *template id*

© 2022 Peter Sommerlad

Class Templates Intro

- Class type definitions can also have template parameters
- since C++17, template argument deduction works for constructors (CTAD) of class templates

Pre C++17 (still possible)

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

Post C++17 (CTAD)

```
std::vector v{1,2,3};
std::vector<int> empty{};
```

for special cases:

- function templates can be ***overloaded***
- class templates can be ***specialized***

© 2022 Peter Sommerlad

Speaker notes

- function templates can be *overloaded* by functions or function templates with the same name for special cases
- class templates can be *specialized* by refining the original class template (see later)

While syntactically it is possible to *specialize* function templates, it is not useful, because in contrast to class templates, the selection of the special cases occurs from a function name's overload set by selecting the best match and only consider function template specialization **after** the generic function template was chosen. Since there is no "overload-resolution" of multiple definitions with the same name for types, class template specializations are considered. Template-specializations always refer to the *primary template*, while function (template) overloads are independent from each other and still considered togehter because of the common name.

© 2022 Peter Sommerlad

Class Template Syntax

```
template<template-parametrs>
struct templatename { /* ... */ };
```

```
template<typename T>
struct Sack { /* ... */ };
```

- class template gives type compile-time parameters
  - members can depend on template parameters
  - member functions are function templates with the class' template parameters
  - you can have member function templates with additional template parameters

© 2022 Peter Sommerlad

Speaker notes

All member functions of a class template must be inline functions defined in the header file together with the class template. This allows the class template to be instantiated later on, like with function templates, the complete code of a template must be visible for the compiler to be able to instantiate the template where it is used.

© 2022 Peter Sommerlad

template class Sack (1)

```
template <typename T>
class Sack
{
  using SackType=std::vector<T>;
  SackType theSack{};
  using size_type=typename SackType::size_type;
  inline static std::mt19937 randengine{};
  using rand=std::uniform_int_distribution<size_type>;
public:
  bool empty() const { return theSack.empty() ; }
  size_type size() const { return theSack.size();}
  void putInto(T const &item) { theSack.push_back(item);}
  template <typename Elt = T>
  explicit operator std::vector<Elt>() const {
    return std::vector<Elt>(theSack.begin(),theSack.end());
              // () to avoid initializer_list ctor
  }

//...
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

template class Sack (2)

```
template <typename T>
class Sack
{
  using SackType=std::vector<T>;
  SackType theSack{};
//...
  T getOut() & {
    using difference_type=typename SackType::difference_type;
      if (empty()) throw std::logic_error{"empty Sack"};
      size_t const index = rand{0u,size() - 1}(randengine);
      T retval{theSack.at(index)};
      theSack.erase(std::next(theSack.begin(),
          static_cast<difference_type>(index)));
      return retval;
  }
};
```


© 2022 Peter Sommerlad

Speaker notes

Member function implementations outside of the class template definition must add the class template introduction in front of the definition, as well as the class template-id in front of the member function name. Since everything must be implemented in a header file anyway, it is useful to just implement everything within the class template (unless too cluttered).

```
class Sack
{
  using SackType=std::vector<T>;
  SackType theSack{};

  using size_type=typename SackType::size_type;
  inline static std::mt19937 randengine{}; // not thread safe
  using rand=std::uniform_int_distribution<size_type>;
```

The random number infrastructure introduced with C++11 is used here. It requires to define a (pseudo-) random number engine that keeps the state between individual random numbers. For the sake of simplicity, this state is kept in a class member variable (**`inline static`** data members are a C++17 feature). Such variables are shared across all objects of a class, but are per-class-template instantiation, since they are implicitly parameterized with the class template parameter(s). However, due to the mutable state, potentially shared across threads, it is not recommended in production code that incorporates multi-threading. See C++Expert for safely dealing with that. instead of manually mapping a random number into the range [`0` .. `size()`) it is better to use the `uniform_int_distribution` that does this correctly.

Note, that the functions dealing with indices in the standard library are inconsistently using unsigned (`size_type`) and signed (`difference_type`) integers.. For example, the indexing `vector::operator[]` is defined over size_type, whereas, indexing via a random-access iterator uses `difference_type` as used within `std::next()`. This can be a curse If you do not turn on corresponding compile warnings, the signed and unsigned integers used will silently convert. However, you might have a value change, if values are negative or very big, which is rare in practice, because you will rarely have an indexable structure with that many elements.

© 2022 Peter Sommerlad

`using` type aliases

```
using SackType=std::vector<T>;
```

- It is common to define short hand aliases
  - less typing and reading
  - single point to adapt a change (DRY principle)
- Aliases can also be templates:

```
template<typename T>
using Vec3 = std::array<T, 3>;
```


© 2022 Peter Sommerlad

`typename` for dependent names (`::`)

```
using size_type=typename SackType::size_type;
using size_type=typename std::vector<T>::size_type;
```

- A name within a template that depends on the template parameter is a “dependent name”
  - `SackType` depends on the template parameter `T`
- Without annotation, the compiler will assume a name that is qualified with a dependent name is no type
  - variable or function
- keyword **`typename`** is required
  - C++20 removed the need when it is obviously a type

© 2022 Peter Sommerlad

Speaker notes

For the using declaration the name `SackType::size_type` the compiler can assume that this is a type, so if the code only needs to be compiled with c++20 that **`typename`** keyword can be omitted.

© 2022 Peter Sommerlad

Example Dependent Names

- Accessing members of a template parameter:

```
template <typename T>
void accessTsMembers() {
  typename T::MemberType m{};
  T::StaticMemberFunction();
  T::StaticMemberVariable;
}
```

```
struct Argument {
  struct MemberType{};
  static void StaticMemberFunction();
  static int StaticMemberVariable;
};
```

Indirect dependency:

```
using size_type=typename std::vector<T>::size_type;
```

- for member class template, need to use `template` if used as a dependent name

© 2022 Peter Sommerlad

Speaker notes

Putting a class template as a member of another class template is possible. However, such uses are considered “expert-level”, therefore, we do not go deeper here.

© 2022 Peter Sommerlad

What is the *concept* of Sack's T

```
template <typename T>
class Sack
{
  using
        SackType=std::vector<T>;

  void putInto(T const &item) {
    theSack.push_back(item);
  }
  T getOut(){
    // ...
    T
        retval{theSack.at(index)};
```


© 2022 Peter Sommerlad

Speaker notes

All the concepts for the template parameter T of our class template `Sack` are implicitly given by the usage of `std::vector<T>`. Note, `std::vector` can be used with types that are move-only, but that requires transfer of ownership (`std::move()`), when using `push_back`

© 2022 Peter Sommerlad

What is the *concept* of Sack's T

```
template <typename T>
class Sack
{
  using
        SackType=std::vector<T>;

  void putInto(T const &item) {
    theSack.push_back(item);
  }
  T getOut(){
    // ...
    T
        retval{theSack.at(index)};
```

- T needs to be destructable (as vector element)

© 2022 Peter Sommerlad

What is the *concept* of Sack's T

```
template <typename T>
class Sack
{
  using
        SackType=std::vector<T>;

  void putInto(T const &item) {
    theSack.push_back(item);
  }
  T getOut(){
    // ...
    T
        retval{theSack.at(index)};
```

- T needs to be destructable (as vector element)
- T needs to be copyable (`push_back`)

© 2022 Peter Sommerlad

What is the *concept* of Sack's T

```
template <typename T>
class Sack
{
  using
        SackType=std::vector<T>;

  void putInto(T const &item) {
    theSack.push_back(item);
  }
  T getOut(){
    // ...
    T
        retval{theSack.at(index)};
```

- T needs to be destructable (as vector element)
- T needs to be copyable (`push_back`)
- T needs to be copy-constructible (in `getOut()`)

© 2022 Peter Sommerlad

Defining members outside of class template

- syntax a bit tedious
- still must be inline (in header)

```
template <typename T>
T Sack<T>::getOut() & {
  using difference_type=typename
        SackType::difference_type;
    if (empty()) throw std::logic_error{"empty Sack"};
    size_t const index = rand{0u,size() - 1}
        (randengine);
    T retval{theSack.at(index)};
    theSack.erase(std::next(theSack.begin(),
        static_cast<difference_type>(index)));
    return retval;
}
```

- repeat template > `template<typename T>`
- member signature > T `Sack<T>::getOut() &`
- including the *template-id* > `Sack<T>::`

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Template Argument Deduction: Factory Function

```
template <typename T>
Sack<T> makeSack(std::initializer_list<T> list){
  Sack<T> sack{};
  for (auto const elt : list){
    sack.putInto(elt);
  }
  return sack;
}
```

We can deduce the Sack's template argument from a ***`std::initializer_list<T>`****'s template argument.*

© 2022 Peter Sommerlad

Speaker notes

Prior to C++17, only function templates allowed to omit template arguments, when they could be deduced from the function call arguments.

To achieve template argument deduction for class template, we need to resort to a function template for deducing the argument(s) of the class template. This is a common pattern for ***factory function templates*** to achieve that.

© 2022 Peter Sommerlad

Class Template Argument Deduction

- defining a `std::initializer_list<T>` constructor

```
explicit Sack(std::initializer_list<T> l):theSack{l}{}
Sack() = default;
```

- allows us to use `Sack` without explicit template argument

```
Sack sack{1,2,3,4,5,6};
```

  - instead of just the factory function template

```
auto sack{makeSack({1,2,3,4,5,6})};
```


© 2022 Peter Sommerlad

Speaker notes

While we still can use factory function templates for deducing the class template arguments, C++17 introduced class template argument deduction, that we already used, when using the C++ standard library.

Note, that we needed to resurrect the default constructor with `Sack()=default;`,that is employed by the factory function template, because defining any other constructor eliminates the compiler-provided default constructor.

© 2022 Peter Sommerlad

Guidelines for Class Templates

- Define class templates completely in header files where the template is defined
- Member functions of class templates
  - Either in class template directly
  - or separate in the same header file
- language elements depending on a template parameter,
  - prefix **`typename`** for a type
  - prefix **`template`** for a member template
- **`static`** member variables of a class template are defined either as **`const`**/**`constexpr`**, or **`inline`**,
  - be aware of mutable data with static storage duration

© 2022 Peter Sommerlad

Speaker notes

`static` (inline or const) member variables of class templates exist per template instantiation, that means per template argument combination for that class template. For example, if a program defines two variables of type `Sack<int>` there exists exactly one variable `Sack<int>::randengine`. Using another type as template argument for Sack, such as `Sack<std::string>` will have its own `static` member variable `Sack<std::string>::randengine`.

`static` member variables of a class template, that are neither const nor inline, require a separate definition outside of the class template, like with `static` members of a regular class. However, such a definition must exist exactly once in a program

© 2022 Peter Sommerlad

Why static members are a bad idea

```
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template<typename T>
struct staticmember {
       inline static int dummy{sizeof(T)};
};
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
```

templatewithstaticmember.h

```
#include "setToDummy.h"
#include "templatewithstaticmember.h"
int foo(){
        using
         dummytype=staticmember<int>;
        dummytype::dummy=42;
        return dummytype::dummy;
}
```

setToDummy.cpp

```
#include "templatewithstaticmember.h"
#include "setToDummy.h"
#include <iostream>
int main(){
        std::cout << staticmember<double>::dummy << '\n';
        std::cout << staticmember<int>::dummy << '\n';
        std::cout << foo() << '\n';
        std::cout << staticmember<int>::dummy << '\n';
}
```

Output (guess):

© 2022 Peter Sommerlad

Speaker notes

Both compilation units share the global variable `staticmember<int>::dummy`

Note: non-const variables with static storage duration are in general a bad idea, because of the issue of data races, when usable from multiple threads.

© 2022 Peter Sommerlad

Why static members are a bad idea

```
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template<typename T>
struct staticmember {
       inline static int dummy{sizeof(T)};
};
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
```

templatewithstaticmember.h

```
#include "setToDummy.h"
#include "templatewithstaticmember.h"
int foo(){
        using
         dummytype=staticmember<int>;
        dummytype::dummy=42;
        return dummytype::dummy;
}
```

setToDummy.cpp

```
#include "templatewithstaticmember.h"
#include "setToDummy.h"
#include <iostream>
int main(){
        std::cout << staticmember<double>::dummy << '\n';
        std::cout << staticmember<int>::dummy << '\n';
        std::cout << foo() << '\n';
        std::cout << staticmember<int>::dummy << '\n';
}
```

Output (guess):

- 8 (platform)

© 2022 Peter Sommerlad

Why static members are a bad idea

```cpp
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template<typename T>
struct staticmember {
        inline static int dummy{sizeof(T)};
};
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
```

templatewithstaticmember.h

```cpp
#include "setToDummy.h"
#include "templatewithstaticmember.h"
int foo(){
        using
         dummytype=staticmember<int>;
        dummytype::dummy=42;
        return dummytype::dummy;
}
```

setToDummy.cpp

```cpp
#include "templatewithstaticmember.h"
#include "setToDummy.h"
#include <iostream>
int main(){
        std::cout << staticmember<double>::dummy << '\n';
        std::cout << staticmember<int>::dummy << '\n';
        std::cout << foo() << '\n';
        std::cout << staticmember<int>::dummy << '\n';
}
```

Output (guess):

- 8 (platform)
- 4 (platform)

© 2022 Peter Sommerlad

Why static members are a bad idea


```cpp
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template<typename T>
struct staticmember {
        inline static int dummy{sizeof(T)};
};
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
```


templatewithstaticmember.h


```cpp
#include "setToDummy.h"
#include "templatewithstaticmember.h"
int foo(){
        using
         dummytype=staticmember<int>;
        dummytype::dummy=42;
        return dummytype::dummy;
}
```


setToDummy.cpp


```cpp
#include "templatewithstaticmember.h"
#include "setToDummy.h"
#include <iostream>
int main(){
        std::cout << staticmember<double>::dummy << '\n';
        std::cout << staticmember<int>::dummy << '\n';
        std::cout << foo() << '\n';
        std::cout << staticmember<int>::dummy << '\n';
}
```


Output (guess):

- 8 (platform)
- 4 (platform)
- 42

© 2022 Peter Sommerlad

Why static members are a bad idea

```
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template<typename T>
struct staticmember {
        inline static int dummy{sizeof(T)};
};
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
```

templatewithstaticmember.h

```
#include "setToDummy.h"
#include "templatewithstaticmember.h"
int foo(){
        using
         dummytype=staticmember<int>;
        dummytype::dummy=42;
        return dummytype::dummy;
}
```

setToDummy.cpp

```
#include "templatewithstaticmember.h"
#include "setToDummy.h"
#include <iostream>
int main(){
        std::cout << staticmember<double>::dummy << '\n';
        std::cout << staticmember<int>::dummy << '\n';
        std::cout << foo() << '\n';
        std::cout << staticmember<int>::dummy << '\n';
}
```

Output (guess):

- 8 (platform)
- 4 (platform)
- 42
- 42

© 2022 Peter Sommerlad

Class Template Gotchas

- Inheritance with templates makes name lookup “interesting”

```
template<typename T>
struct parent {
  int foo() const {
    return 42;
  }
  static int const bar { 43 };
};
int foo() {
  return 1;
}
double const bar { 3.14 };
```

```
template<typename T>
struct gotchas: parent<T> {
  std::string demo() const {
    std::ostringstream result { };
    result << bar << " bar \n";
    result << this->bar << " this->bar \n";
    result << gotchas::bar << " gotchas::bar\n";
    result << foo() << " foo() \n";
    result << this->foo() << " this->foo() \n";
    return result.str();
  }
```

What is the content of result of `gotchas<int>{}.demo()`?

https://godbolt.org/z/8fKM1rhbf

© 2022 Peter Sommerlad

Speaker notes

https://godbolt.org/z/8fKM1rhbf

“3.14 bar ” “43 this->bar ” “43 demogotchas::bar” “1 foo() ” “42 this->foo() ”

When a template inherits from a base class that is dependent on its template parameter(s), then name lookup will not assume that it knows the content of the base class, because the concrete base class might actually be a specialization. This means, to address a name that is (should be) defined in the base class template, one needs to prefix the name with either `this->` or with the class name or the base-class template id.

© 2022 Peter Sommerlad

Class Templates that inherit from a dependent base

- Rule: always use `this->` or the class `name::` to refer inherited members in a class template

```
this->bar        gotchas::bar
```

- If the name could be a *dependent name* it will not be searched for when compiling the template
- checks are only made for dependent names when a template is instantiated
  - failing those checks can lead to the unwieldy long error messages in nested template instantiations “… instantiated from…”

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Unwanted Template Arguments

- `Sack<T>` could be instantiated with a Relation type (but not with a reference type)
  - would require all Subjects referred to need to outlive the usage of the Sack
  - also someone needs to clean up those subjects afterwards
  - pointer types are culprits of such dangers
- prohibit pointer types to instantiate with:
  `Sack<int *> shouldNotCompile{};`
- `Sack drawnames{"Peter", "Toni", "Thomas", "Georg", "Michael"};` might be nice

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Class Template Specialization

- **partial** with template paramter(s)
- full **explicit specialization** with a full set of arguments
- non-specialized template must be declared first
- the best-fit, **most-specialized** version is chosen
- must be in same namespace than **primary template**

```
template <typename T> class Sack; // forward declaration
```

partial for pointers

```
template <typename
          T>
struct Sack<T*>
{
  ~Sack()=delete;
};
```

full for string literals

```
template <>
struct Sack<char const *>
: Sack<std::string> {
  using Sack<std::string>::Sack;
};
```


© 2022 Peter Sommerlad

Speaker notes

Instead of "overloads" like for function templates, class templates are specialized for letting the compiler select a best match

Note that a specialization will list the template arguments after the original template's name within angle brackets `Sack<T *>`. There are several options to prohibit instantiating a template. One option shown here, is to delete the destructor, which also prevents creating the template. As you can see, a template specialization is completely detached from the original template with respect to its defined members. Another option would be to just declare the template specialization without providing a definition:

```
template <typename T>
struct Sack<T*>;
```

Both will lead to compile errors (different ones), when an instantiation of `Sack<T>` with a pointer type for T is attempted.

Trying to create a `Sack<int*>` with the deleted destructor we get: "error: attempt to use a deleted function"

The missing definition results in: "error: implicit instantiation of undefined template 'Sack<int *>'"

A full explicit specialization still needs the lead in with **`template<>`** even if no more parameters are needed. For function templates, one would use a regular function overload with the name of the function template instead.

The specialization for `char const *` which is the decayed type of string literals, we inherit from Sack to actually get the behavior of a `Sack` that contains strings. This eliminates the potential lifetime problem that holding a raw pointer in the underlying vector would have had.

C++20 introduced concepts as a syntactical means to restrict template instantiations and direct specialization selection, which we will go into much more detail in the C++Expert course.

© 2022 Peter Sommerlad

Specializations are independent

```
template <>
struct Sack<char const *> {
  using SackType = std::vector<std::string>;
  using size_type = SackType::size_type;
  SackType theSack;
public:
  bool empty() const {
    return theSack.empty();
  }
  size_type size() const {
    return theSack.size();
  }
  void putInto(char const *item) { theSack.push_back(item);}
  std::string getOut() {
    if (empty()) throw std::logic_error{"empty Sack"};
    std::string result=theSack.back();
    theSack.pop_back();
    return result;
  }
};
```


© 2022 Peter Sommerlad

Speaker notes

An alternative specialization for `Sack<char const *>` is shown here. It has different semantics of `getOut()`. There is absolutely no check by the compiler if the specialization and the primary template actually match, with respect to their provided members. Only the actual use of a member will trigger this member's instantiation. This allows to partially fulfil a template parameter's concept, if the unfulfilled part is never needed by the instantiated members.

© 2022 Peter Sommerlad

More Sack constructors

- already seen: `std::initializer_list` constructor

```
Sack<char> charSack{'a', 'b', 'c'};
```

more like `std::vector` provides:

- fill a `Sack<T>` from a range given by two iterators

```
std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
Sack<int> aSack{begin(values), end(values)};
```

- Create a `Sack<T>` of multiple default values

```
Sack<unsigned> aSack(10u, 3u);// 10 time value 3u
```


© 2022 Peter Sommerlad

Speaker notes

Providing additional constructors in addition to a constructor taking a `std::initializer_list<T>` raises the problem of disambiguation of constructor calls by using round parentheses `()` for all the non-`inializer_list` constructors.

© 2022 Peter Sommerlad

Sack from iterator-pair

simple delegation to `std::vector` constructor

```
Sack()=default;
template <typename ITER>
Sack(ITER b, ITER e):theSack(b,e) {}
Sack(std::initializer_list<T> il):theSack(il) {}
```

This is not perfect, because it could also be used with two integers for a `Sack<int>`.

© 2022 Peter Sommerlad

Speaker notes

The constructor template can be used for delegation to both two argument constructors with homogeneous parameter types of `std::vector`, even if it is only meant to be used with iterators.

In the C++ Expert course we will learn more on how to constrain templates, so that only arguments with given properties are allowed. This can be achieved with C++20 concepts and requires clauses, or pre-C++20 with a mechanism with a bit uglier syntax that causes SFINAE (substitution-failure is not an error) for unwanted cases.

© 2022 Peter Sommerlad

More Sack show-off features

- Obtain a copy of contents in a `std::vector` (for inspection)

```
Sack<int> aSack{1, 2, 3};
auto values { static_cast<std::vector<int>>(aSack)
        };
auto dvalues { aSack.asVector<double>() };
```

  - Deducing T from initializer list or range

```
Sack intSack{1, 2, 3};
Sack aSack(begin(values), end(values));
```

  - Specify the kind of container template to be used

```
Sack<int, std::set> aSack{1, 2, 3};
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Content as `std::vector`

show member template (operator overload)

```
template <typename Elt>
explicit operator std::vector<Elt>() const {
  return std::vector<Elt>{begin(theSack),end(theSack)};
}
template <typename Elt=T>
std::vector<Elt> asVector() const {
  return std::vector<Elt>{begin(theSack),end(theSack)};
}
```


© 2022 Peter Sommerlad

Speaker notes

In general, I do not recommend to implement conversion operators, especially not as template. This is just to demonstrate the syntax and abilities of it.

Ensure to make conversion operators **`explicit`**!

The named extraction function `asVector<Elt>` might be a better solution and is actually easier to call than the conversion operator.

As long as the Sack template parameter type `T` is convertible to the element type `Elt` of the vector to extract values from, it is possible to obtain a vector with a different element type than the element type of the Sack.

© 2022 Peter Sommerlad

Deduction of Sack template argument

```
  Sack(std::initializer_list<T> il):theSack(il) {}
```

```
void deductionFromInitializerList(){
  Sack anIntSack{1,2,3,4,5};
  ASSERT_EQUAL(5,anIntSack.size());
}
```

- Constructor parameter depends on template parameter
- Implicit ***deduction guide*** will be able to deduce T

© 2022 Peter Sommerlad

Speaker notes

You cannot suppress implicit deduction guides, but if the deduction of the template argument of a class template is not as obvious, one can specify a separate “explicit deduction guide” that looks like a bit function template declaration with the “constructor call” in front of `->` and the chosen class template following.

© 2022 Peter Sommerlad

Explicit Deduction Guides

- User-defined deduction guides can be specified in the same scope as their class template
  - usually just after the template definition

`ClassTemplateName(ConstructorParameters) ->TemplateId;`

- explicit deduction guide:

```
template<typename ITER, template<typename ...> typename
        container=std::vector>
Sack(ITER,ITER)->Sack<typename std::iterator_traits<ITER>::value_type,
        container>;
```


© 2022 Peter Sommerlad

Speaker notes

Cevelop does not yet understand the C++17 syntax of deduction guides. So do not be confused by the wiggly lines in Cevelop. Code should just compile fine.

The `std::iterator_traits` class template acts as a “meta-function” to compute another type from the given iterator type.

A deduction guide is used to deduce the Sack’s template parameter from those constraint constructors, i.e., by selecting the iterator’s value_type automatically. However, this is tricky, because a naïve attempt, easily can cause crashing code:

```
void creationFromIteratorsCanDeduceElementType(){
  std::vector<std::string> v(2,"Hello");
  Sack asack(begin(v),end(v));
  ASSERT_EQUAL("Hello",asack.getOut());
}
void creationFromStringLiteralsSelectsWrongSackType(){
  // Sack("A","B") -> Sack<char> because of deduction guide
  static_assert(std::is_same_v<Sack<char>,decltype(Sack("A","B"))>);
}
```

That last test case only has a compile-time check, because initializing a Sack from two string literals will cause a crash (best case) or undefined behavior. because the pointer decay of the literals forms an invalid range.

© 2022 Peter Sommerlad

Constructor for N-Elements

deduction guide limits does not allow the use of our 2 parameter constructor

```
Sack<std::string> asack(10u, "hello");
ASSERT_EQUAL(10, asack.size());
ASSERT_EQUAL("hello", asack.getOut());
```

constructor for n elements:

```
Sack(size_type n, T elt):theSack(n,elt) {}
```

Is an explicit deduction guide needed for this constructor?

© 2022 Peter Sommerlad

Speaker notes

There is no need to define an explicit deduction guide, because the constructor is directly dependent on the class template parameter.

© 2022 Peter Sommerlad

Varying Sack's Container

- std::vector might not be perfect, because of the "in-the-middle" erase

```
Sack<int, std::set> aSack{1,2,3};
```

  - need to ensure `getOut()` is not vector-specific

```
T getOut() {
  if (empty()) throw std::logic_error {"empty Sack"};
  difference_type index = rand {0,static_cast<difference_type>(size()-1)}(randengine);
  auto iter=std::next(begin(theSack),index);
  T retval {*iter};
  theSack.erase(iter);
  return retval;
}
```


© 2022 Peter Sommerlad

Speaker notes

We use `std::next` instead of direct element access with `vector::at(index)` to advance the iterator to the position we want to obtain and erase the element.

© 2022 Peter Sommerlad

Template Template Parameter (1)

How can we vary the underlying container?

```
template<typename T, template<typename> typename container>
class Sack;
```

- assumes single argument template for container
- problem: standard container have more optional template parameters

```
Sack<int, std::set> aSack{1, 2, 3}; //Does not compile
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Variadic Template Template Parameter

```
template<typename T, template<typename ...> typename container>
class Sack;
```

- allow for a class template argument with any number of parameters
- variadic templates only make that useful

```
Sack<int, std::set> aSack{1, 2, 3}; // works now
```


© 2022 Peter Sommerlad

`std::vector` as default container

- previous declaration requires changing all existing code
  - template template parameter must be specified
- default template template argument as solution:

```
template<typename T, template<typename ...> typename container=std::vector>
class Sack
{
  using SackType=container<T>;
  SackType theSack {};


  Sack<int,std::list> listsack{1,2,3,4,5};
```


© 2022 Peter Sommerlad

Speaker notes

Since we cannot deduce the template template argument (unless it is using the default), we require another factory function that can do so. This factory function takes the template template parameter first

© 2022 Peter Sommerlad

Element Type Deduction with template-template Parameter

- Factory Function makeSetSack()

```
template<typename T>
auto makeSetSack(std::initializer_list<T> list) {
  return Sack<T, std::set>(begin(list), end(list));
}
template<typename ITER>
auto makeSetSack(ITER b, ITER e) {
  return Sack<typename std::iterator_traits<ITER>::value_type, std::set>(b, e);
}
```

- CTAD for Alias Templates (C++20)

```
template<typename T>
using SetSack = Sack<T,std::set>;


  SetSack setsack{{1,2,3,4,5}}; // implicit deduction guide OK
```


© 2022 Peter Sommerlad

Speaker notes

Unfortunately, I could not make CTAD work for an alias template with the explicit deduction guide. This might be due to incomplete implementation of C++20 features.

So for initializing a SetSack from a pair of iterators, one needs to either specify the element type as template argument, or use the corresponding factory function.

```
void makeSetSackFromIterators(){
  std::vector v{1,2,3,4};
  auto setsack(makeSetSack(begin(v),end(v)));
  static_assert(std::is_same_v<Sack<int,std::set>,decltype(setsack)>);
  ASSERT_EQUAL(4, setsack.size());
  ASSERT_EQUAL((std::vector{1,2,3,4}),setsack.asVector());
}

void SetSackAliasCTAD(){
  SetSack setsack{{1,2,3,4,5}}; // implicit deduction guide OK
  static_assert(std::is_same_v<Sack<int,std::set>,decltype(setsack)>);
  ASSERT_EQUAL(5, setsack.size());
  ASSERT_EQUAL((std::vector{1,2,3,4,5}),setsack.asVector());
}

void SetSackAliasCTADFromIteratorsDoesNotWork(){
  std::vector v{1,2,3,4};
  SetSack<int> setsack(begin(v),end(v)); // must specify template argument
  static_assert(std::is_same_v<Sack<int,std::set>,decltype(setsack)>);
  ASSERT_EQUAL(4, setsack.size());
  ASSERT_EQUAL((std::vector{1,2,3,4}),setsack.asVector());
}
```

© 2022 Peter Sommerlad

Adapting standard Containers

Inheriting from standard library classes is not officially sanctioned by the C++ standard

- possible to wrap as member (like Sack)
  - need to add similar constructors
  - limit/adapt functionality
- as base class
  - can use “inheriting constructors”
  - replace or extend functionality
  - no polymorphism
  - might hide base overloads accidentally

© 2022 Peter Sommerlad

Speaker notes

Inheriting from standard library containers does not provide substitutability like with polymorphic base classes, but it allows to adapt these containers with own functionality and to replace implementations, as long as one does not call base class members directly or relies on implicit base conversions.

© 2022 Peter Sommerlad

Example: `SafeVector<T>`

std::vector speed risks undefined behavior
`v[i]`, `v.front()`, `v.back()` are risky

```
template<typename T>
struct safeVector:std::vector<T> {
using std::vector<T>::vector;
using size_type=typename std::vector<T>::size_type;
decltype(auto) operator[](size_type i) & { // or T&
        return this->at(i);
}
decltype(auto) operator[](size_type i) const & { // or T const &
        return this->at(i);
}
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

`SafeVector<T>` Deduction Guides

Inheriting constructors does provide deduction guides…

```
// needs deduction guide
template<typename ITER>
safeVector(ITER,ITER) -> safeVector<typename
        std::iterator_traits<ITER>::value_type>;
template<typename T>
safeVector(std::initializer_list<T>) -> safeVector<T>;
template<typename T>
safeVector(typename safeVector<T>::size_type,T) -> safeVector<T>;



  safeVector v{1,2,3};
  v.clear();
  ASSERT_THROWS(v.front(),std::out_of_range);
}
```


© 2022 Peter Sommerlad

Speaker notes

While C++20 defined “inheriting” deduction guides for alias template, the inheritance of deduction guides for base class templates with inheriting constructors was not introduced with C++20 due to time constraints. As of today, the needed wording never appeared, so the feature fell between the cracks, even for C++23, it seems.

© 2022 Peter Sommerlad

Exercise 5 ()

https://github.com/PeterSommerlad/CPPCourseAdvanced/

© 2022 Peter Sommerlad

Variable Templates

and Meta-Programming Introduction

- similar to class templates
- partial and full specialization is possible
- variable templates are actually compile-time constants
- useful for defining concepts = type properties
  - and vice versa for using `require`

© 2022 Peter Sommerlad

Speaker notes

We won't be able to go into full features meta programming (code that creates other code) in this course. However, a few tricks to rely on for "normal" templates, to ensure that things that compile actually achieve what one expects and things that should not compile actually don't.

© 2022 Peter Sommerlad

<type_traits>

- many type properties are computable through variable templates
- can be ensured via **static_assert();**

```
template<typename T>
  struct Sack{
    static_assert(not std::is_pointer_v<T>);
  };
```

- type traits variable templates end in _v

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Specifying Variable Templates

```
template <TemplateParameters>
constexprType TemplateName{Initializer} ;
```

- simple form (can also use = for init)

```
template<typename T>
constexpr T pi = T(3.14159265358979323851);
```

- allows

```
auto const circumference{ diameter * pi<double>};
float const area{ radius * radius * pi<float>};
auto justthree { pi<int> }; // bad
```


© 2022 Peter Sommerlad

Speaker notes

© 2022 Peter Sommerlad

Variable Templates for Traits

```
template<typename T>
constexpr bool isPointer { false };
template<typename T>
constexpr bool isPointer<T*> { true };
template<>
constexpr bool isPointer<void*> { false };
```

- like class templates:
  - primary template
  - partial specialization
  - full specialization
- for different behavior

© 2022 Peter Sommerlad

Speaker notes

Note that the standard library provides traits to determine if a type is a pointer. This is just for illustration purposes.

```
static_assert(not isPointer<int>);
static_assert(isPointer<int*>);
static_assert(isPointer<int const *>);
static_assert(not isPointer<void*>);
static_assert(isPointer<void const *>); // oops specialization must match exactly
```

© 2022 Peter Sommerlad

C++20 Variable Templates from Concepts

- `require` expression allows to compute initial value but is not always perfect

```
template<typename T>
constexpr bool
isDereferencable {
  requires(T t){  *t; }
};
```

```
static_assert(not isDereferencable<int>);
static_assert(    isDereferencable<int *>);
static_assert(    isDereferencable<int (*)()>); // function pointer
static_assert(not isDereferencable<void *>);
```


© 2022 Peter Sommerlad

Speaker notes

It is not desirable to consider `void*` or function pointers to be dereferencable, because the result is not a value that can be used.

© 2022 Peter Sommerlad

Checking for result of `*t`

- compound requirements can check if result type of expression conforms to a concept

```
template<typename T>
constexpr bool
isDereferencable {
  requires(T t){
    { *t } -> IsObject;
  }
};
```

  - need to define corresponding concept:

```
template<typename T>
concept IsObject =
    std::is_object_v<std::remove_reference_t<T>>;
```

- `std::remove_reference_t<T>` is a “meta-function” computing a type from `T`

© 2022 Peter Sommerlad

Speaker notes

Type-“meta-”functions are template (aliases) that take a type parameter and have a type as a result. This is pure compile-time computation without any run-time overhead. You can see more on such meta programming in C++ Expert course.

However, there are books just on C++ 17 Templates with almost 800 pages, so do not expect to learn everything possible in a few course days.

And within the C++ language standard the section on templates is one of the hardes to understand, even so, it contains some deliberate fun (ISO 14882 C++ [temp.expl.spec] p.8(C++20)/p.7(C++17)):

“The placement of explicit specialization declarations for function templates, class templates, variable templates, member functions of class templates, static data members of class templates, member classes of class templates, member enumerations of class templates, member class templates of class templates, member function templates of class templates, static data member templates of class templates, member functions of member templates of class templates, member functions of member templates of non-template classes, static data member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, variable templates, member class templates of non-template classes, static data member templates of non-template classes, member class templates of class templates, etc., can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. **When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.**”

© 2022 Peter Sommerlad

Summary Variable Templates

- Use for compile-time computation
- type-specific constants (`pi<long double>`)
- type traits via bool variable templates and specializations
- type-trait as concept implementation
- concept as type-trait implementation
- **`static_assert`** with type traits to prevent unwanted instantiations
- template aliases as type-meta-functions

© 2022 Peter Sommerlad

Speaker notes

C++ templates allow programming with types at compile time, like one would use values at run-time. If one gets over the angle bracket syntax, things start to make sense for a programmer used to functional languages. Variable templates, concepts and type meta-functions allow sophisticated type safe and optimized programming, for beyond what we can do now. However, there is great potential for unintelligible error messages and subtle things not working as naïvely expected, because of little details, i.e., `void const` is not the same as `void` or `*t` resulting in a reference (lvalue), not a value type.

© 2022 Peter Sommerlad

Exercise 6

exercises/exercise06.md

© 2022 Peter Sommerlad

Speaker notes

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise06.md

© 2022 Peter Sommerlad

Storage

- Automatic - local
- Static - global
- Dynamic - heap

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Automatic vs. Static vs. Dynamic

Different Memory Regions and Storage Duration

- local variables - stack
  - “automatic” storage duration
  - clean up at scope closing }
  - tends to be limited
- global variables - bss
  - “static” storage duration
  - clean up at program exit
  - danger of UB on construction and cleanup
- heap
  - “dynamic” storage duration
  - create and destroy programmatically
  - requires managing

© 2022 Peter Sommerlad

Speaker notes

While we learned to avoid global variables (unless const), space for local variables can be a limiting factor.

On linux or macos you can try the command `ulimit -a` and see if it outputs a limit for stack size among other things. For example, on my Mac the stack size is limited to 8 MB. One reason such a limit is imposed is for the operating system to detect endless recursion early enough to stop a program before it uses up so many resources, that the system is overwhelmed.

Even so, we didn't do so explicitly, we used types like std::vector that employ dynamic memory that is

© 2022 Peter Sommerlad

When to use dynamic memory

- Overcome limited local variable space
  - `std::vector`
  - "optimize" copying big object's
- "linked" (recursive) object structures
  - linked lists (`std::list`)
  - trees (`std::set`)
  - other object graphs
  - some Design Patterns (e.g. Decorator, Composite)
- polymorphic factory functions
  - return base-class "pointer" to derived class object

© 2022 Peter Sommerlad

Speaker notes

In the past, dynamic memory was often used to prevent returning “large” object from functions, because the cost of additional copying was considered significant.

However, C++17 made returning temporary objects mandatory without copying and explicitly allows to optimize returning a local variable (named-return-value optimization NRVO).

© 2022 Peter Sommerlad

Legacy Uses of Heap

Do not do this in your code!

```
auto ptr = new int{42}; // acquire dynamic memory
std::cout << *ptr << '\n';
delete ptr; // release dynamic memory
```

- Dangers: memory leaks, dangling, double delete

*Refactor existing code using **`new`** or **`delete`***

© 2022 Peter Sommerlad

Speaker notes

C++ does not provide the automatic garbage collection style of languages like Java. However, the deterministic object lifetime model and destructors provide means to use the heap safely.

Even if the new and delete expressions occur in local scope, this code style can produce memory leaks, if the code in between throws an exception. And you might not be able to know if any function called might actually throw, unless there is guaranteed no undefined behavior and all functions called (directly or indirectly) are marked as **noexcept**

© 2022 Peter Sommerlad

Modern way of Heap usage

- Don't use dynamic memory explicitly, rely on existing manager types `std::vector` etc.
- If you must, use `std::make_unique<T>()`:

```
{
  auto ptr = std::make_unique<int>(42); // uses new
  std::cout << *ptr << '\n';
} // ptr is releasing heap memory
```

`std::unique_ptr<T>` is a generic unique manager for dynamic memory `#include <memory>`

© 2022 Peter Sommerlad

Speaker notes

The example given is still ridiculous, because it makes no sense to put an integer into dynamic memory.

© 2022 Peter Sommerlad

std::unique_ptr<T>

- big local data that could cause stack overflow
- sole ownership guaranteed
  - can be returned from a factory function
  - pass by reference of the managed object

```
void foo(large &);
auto p { make_unqiue<large>()};
if (p) { foo(*p); }
```

- or pass to a function transferring ownership

```
void foo(unique_ptr<T>);
auto p { make_unqiue<large>()};
foo(std::move(p)); // p is nullptr afterwards
```


© 2022 Peter Sommerlad

Speaker notes

In the first case, accessing the managed object works through the overloaded `unique_ptr::operator*()`.

Typically, when accessing the managed object in dynamic memory, it is desirable to check if the unique_ptr actually refers to such an object:

© 2022 Peter Sommerlad

Transfer of Ownership

```
std::cout << std::boolalpha;
auto pi = std::make_unique<int>(42);
std::cout << "*pi = " << *pi << '\n';
std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n';
auto const pj = std::move(pi);
std::cout << "*pj = " << *pj << '\n';
std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n';
```

*it is ridiculous to use **`unique_ptr`** for int !!!*

© 2022 Peter Sommerlad

Speaker notes

the explicit conversion to bool is usually used in a conditional and then does not require the static_cast

`if (up) { std::cout << *up; }` is what is used for example.

The example code lacks a check that make_unique actually could acquire heap memory for the object. In resource constraint environments this can fail and the resulting `unique_ptr` is equivalent to **`nullptr`**

Output:

```
*pi = 42
pi.valid? true
*pj = 42
pi.valid? false
```

`std::move(p)` is required, when a `unique_ptr` p needs to transfer ownership. It is not needed, for example, when the result of `std::make_unique()` is used as a function argument, where the function expects a `std::unique_ptr` value.

© 2022 Peter Sommerlad

`unique_ptr` for polymorphic factories

```
std::unique_ptr<std::ostream>
os_factory(std::string const filename) {
  using namespace std; // for shorter code
  if (filename.size()) {
    return make_unique<ofstream>(filename);
  } else {
    return make_unique<ostringstream>();
  }
}
int main(){
  auto out = os_factory("");
  if (out) {
    (*out) << "Hello world\n";
  }
  auto fileout = os_factory("hello.txt");
  if (fileout) {
    (*fileout) << "Hello, world!\n";
  }
}
```


© 2022 Peter Sommerlad

Speaker notes

Note, we could return by value, but not for a polymorphic result referring the base class. Returning a reference would dangle immediately.

© 2022 Peter Sommerlad

`unique_ptr` for legacy C code(1)

```
struct free_deleter {
  template<typename T>
  void operator()(T * p) const {
    std::free(const_cast<std::remove_const_t<T> *>(p));
  }
};
template<typename T>
using unique_C_ptr = std::unique_ptr<T, free_deleter>;
std::string plain_demangle(char const * name) {
  unique_C_ptr<char const> toBeFreed{__cxxabiv1::__cxa_demangle(name, 0, 0,
        0)};
  std::string result(toBeFreed.get());
  return result;
}
```


© 2022 Peter Sommerlad

Speaker notes

An alternative to provide the deleter class type by defining it, could be to use a lambda:

```
std::string demangle(std::string name){
  if (name.size() > 0 ){
    constexpr auto freemem{ [](char *toBeFreed){
        std::free(toBeFreed);
    }};
    std::unique_ptr<char ,decltype(freemem)> freeit {
        abi::__cxa_demangle(name.c_str(),0,0,0),
        freemem };
        std::string result{freeit.get()};
        return result;
  } else {
        return "unknown";
  }
}
```

© 2022 Peter Sommerlad

Guidelines `std::unique_ptr<T>`

- As member variable:
  - For polymorphic reference `std::unique_ptr<base>`
- As local variable:
  - To implement RAII as a (generic) unique manager type for “pointer-like” resources references
  - for very very large objects
  - `std::unique_ptr<T> const p{ std::make_unique<T>(...)};`
    - cannot transfer ownership
    - cannot leak

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

`unique_ptr<T[]>`

- `unique_ptr` can be used for runtime-sized arrays
- prefer `std::vector` instead! -> it just works
  - if you want to prevent expanding over initial size, wrap `std::vector`.
- C++ Expert trainging will demonstrate using `unique_ptr<T[]>`

© 2022 Peter Sommerlad

Speaker notes

`unique_ptr` for arrays of values with a fixed run-time size is possible as a building block for creating your own container types. In general it is better to just use `std::vector`.

© 2022 Peter Sommerlad

`std::shared_ptr<T>`

- `std::unique_ptr` allows only one owner, no copy
- `T&` prevents assignment as a member and can dangle
- `std::shared_ptr` works like Java/Python references
  - copying is possible, referring to the same object
  - re-assignment is possible
  - last reference ceasing to exist, deletes referred object
- factory: `std::make_shared<T>()`
  - all arguments that T's constructors take

© 2022 Peter Sommerlad

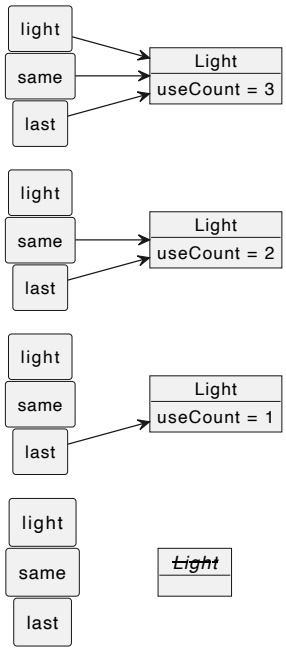
Speaker notes

If this sounds like you want to use `shared_ptr` everywhere, be warned that it incurs significant overhead and one can still create memory leaks, with circular structures

© 2022 Peter Sommerlad

`shared_ptr` mechanics

```
struct Light {
  Light(std::ostream & out) : out{out} {
    out << "Turn on\n";  }
  ~Light() { out << "Turn off\n"; }
  std::ostream & out;
};
int main() {
  auto light{std::make_shared<Light>(std::cout)};
  auto same{light};
  auto last{same}; // 3
  light.reset();   // 2
  same.reset();    // 1
  last.reset();    // deleted
}
```



© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

`shared_ptr` Cycles

```
#include <memory>
using HalfElfPtr = std::shared_ptr<struct HalfElf>;
struct HalfElf {
  explicit HalfElf(std::string name) : name{name}{}
  std::string name{};
  std::vector<HalfElfPtr> siblings{};
  // demo only:
  ~HalfElf(){ std::cout << name << " killed\n"; }
};
void middleEarth() {
  auto elrond = std::make_shared<HalfElf>("Elrond");
  auto elros = std::make_shared<HalfElf>("Elros");
  elrond->siblings.push_back(elros);
  elros->siblings.push_back(elrond);
}
```

https://godbolt.org/z/Ms8n6vssM

© 2022 Peter Sommerlad

Speaker notes

play with it (if you remove one sibling association, then there is no longer a circle):

https://godbolt.org/z/Ms8n6vssM

`-fsanitize=address` as compiler option can help to find such leaks.

Using `std::weak_ptr` it is possible to break such shared_ptr circles, but this still can result in memory not released and requires care to not lose a `shared_ptr` prematurely. At least `std::weak_ptr` cannot dangle as would be the case with a raw pointer.

© 2022 Peter Sommerlad

`shared_ptr` effects

- last shared_ptr handle destroyed deletes allocated object
- instances created with `make_shared<derived>()` and stored in a `shared_ptr<base>` does not need to have a virtual destructor in base
  - `shared_ptr<derived>` memorizes the derived class' destructor and thus `shared_ptr<base>` employs correct destruction code.
- `shared_ptr` can lead to object cycles that persist, even if the program no longer can access them
  - `std::weak_ptr` can break such cycles

© 2022 Peter Sommerlad

Speaker notes

`shared_ptr<derived>` memorizes the class' destructor to be called and thus `shared_ptr<base>` can employ the correct destruction code.

© 2022 Peter Sommerlad

Use cases for `shared_ptr`

- creating a graph of nodes
  - but consider using a library for that
- run-time polymorphic variables with lifetime guarantee
  - if you cannot control lifetime via call hierarchy
- factory result for heap allocated objects (better `make_unique`!)
- only if really best choice:
  - (const) references as parameter or members don't work
  - value type or containers with value types don't work
  - unique_ptr is insufficient

© 2022 Peter Sommerlad

Speaker notes

`shared_ptr<T>` incurs a significant run-time overhead, not only for the heap allocating, but also when passing around, because the reference-counting mechanic requires thread synchronisation (atomic counter).

While I was a big fan of `shared_ptr` 10+ years ago, I learned to understand its overhead. It can even lead to very subtle memory leaks (memory not released), even when the managed object is destroyed.

While the reference count managed by shared_ptr is synchronized across thread usages, the managed object itself is not. One useful approach ist to used `std::shared_ptr<T const>` in a sharing across thread situation.

© 2022 Peter Sommerlad

Object relationships

- Create a Person class that can represent its parents and children
- If a person dies, it ceases to be a a child of its parents and a parent of its children

© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Cannot nest with values

- You cannot use member variables with the type of the class they are defined in.
  - endless recursion
  - technically, class is incomplete, until `};`

```
struct Matryoshka {
Matryoshka nested; // does not compile
};
```

- this needs a relation, e.g. `shared_ptr`, not a value:

```
struct Matryoshka {
  std::shared_ptr<Matryoshka> nested;
};
```


© 2022 Peter Sommerlad

Speaker notes

Other relation types are possible but result in different drawbacks:

- `Matryoshka &` - must be initialized, cannot be empty, cannot reassign
- `Matryoshka *` - must be initialized, can be **`nullptr`**, can dangle
- `std::reference_wrapper<Matryoshka>` - must be initialized, cannot be empty, allows assignment/rebinding
- `std::unique_ptr<Matryoshka>` - cannot dangle, cannot be shared (unless via raw pointer)

© 2022 Peter Sommerlad

Break shared ptr cycles

- parent-child cycle needs to be broken
- `std::weak_ptr<Person>` refers to a `shared_ptr<Person>`
  - re-acquire `shared_ptr<Person>` with `.lock()`
  - need to check if `shared_ptr` is valid

```
{.cpp style=" margin: 0 0 0 0"}
struct Person {   std::string name{};
std::shared_ptr<Person> child{};
std::weak_ptr<Person> parent{};
Person(std::string n):name{n}{} };
```


© 2022 Peter Sommerlad

Speaker notes

We need to define a constructor, because make_shared cannot use aggregate initialization (up to C++20, where it should work, but is not yet implemented on all compilers.

© 2022 Peter Sommerlad

break cycles with `weak_ptr`

```
{
  auto anakin = std::make_shared<Person>("darth vader");
  auto luke = std::make_shared<Person>("luke skywalker");
  anakin->child = luke;
  luke->parent = anakin;

  anakin.reset();
  ASSERT(luke->parent.expired());
  ASSERT_EQUAL(1,luke.use_count());
} // no leak here!
```


© 2022 Peter Sommerlad

Speaker notes

Since there is only one shared_ptr to anakin, “forgetting it” actually destroys the “darth vader” Person.

© 2022 Peter Sommerlad

Access through `weak_ptr`

```
friend
std::ostream &operator<<(std::ostream &os, Person const
      &p){
  return os<<p.name;
}
void acquireMoney(std::ostream &out){
  if (auto realparent = parent.lock()){
    out << "borrow from parent " << *realparent << "\n";
  } else {
    out << "go to bank instead\n";
  }
}
```

```
auto anakin = std::make_shared<Person>("darth vader");
auto luke = std::make_shared<Person>("luke
      skywalker");
anakin->child = luke;
luke->parent = anakin;
std::ostringstream os{};
luke->acquireMoney(os);
ASSERT_EQUAL("borrow from parent darth
      vader\n",os.str());
anakin.reset(); // kill creditor
os.str(""); // reset tracks
luke->acquireMoney(os);
ASSERT_EQUAL("go to bank instead\n",os.str());
```


© 2022 Peter Sommerlad

Speaker notes

for your own notes

© 2022 Peter Sommerlad

Spawning children?

```
auto spawn(std::string name){
  child = std::make_shared<Person>(name);
  // how to set myself as parent?
  return child;
}
```

```
struct Person : std::enable_shared_from_this<Person> {
  std::shared_ptr<Person> child{};
  std::weak_ptr<Person> parent{};
  std::string name{};

  Person(std::string n):name{n}{}
  auto spawn(std::string name){
    child = std::make_shared<Person>(name);
    child->parent = this->weak_from_this();
    return child;
  }
};
```


© 2022 Peter Sommerlad

Speaker notes

Passing the class to a base-class template as the template argument is a case of the **Curious Recurring Template Parameter Pattern** (CRTP Pattern).

`std::enable_shared_from_this<Person>` adds a weak pointer to the object, that is automatically initialized by `make_shared<Person>` The member function `this->weak_from_this()` provides access to this weak pointer. Analogous the function `this->shared_from_this()` implicitly calls `.lock()` on the stored weak pointer, to obtain the associated `shared_ptr`. If the object under consideration is not allocated on the heap, these pointers are equivalent to the `nullptr`, even so, `this` is not a nullptr.

```
void NonHeapObjectHasNullptrWeakFromThis(){
  Person p{"notOnHeap"};
  ASSERT_EQUAL(0,p.weak_from_this().use_count());
  ASSERT(p.weak_from_this().expired());
  ASSERT_THROWS(p.shared_from_this(), std::bad_weak_ptr);
}
```

© 2022 Peter Sommerlad

Multiple Children ?

```
struct Person : std::enable_shared_from_this<Person> {
  std::string name{};
  std::vector<PersonPtr> children{};
  std::weak_ptr<Person> parent{};

  Person(std::string n):name{n}{}
  auto spawn(std::string name){
    auto child = std::make_shared<Person>(name);
    child->parent = this->weak_from_this();
    children.push_back(child);
    return child;
  }
```

```
bool isChildOf(PersonPtr person) const {
  auto myparent = parent.lock();
  if (myparent && person){
    if (myparent->name == person->name){
      // candidate:
      auto me = this->shared_from_this();
      return std::any_of(begin(person->children)
             , end(person->children)
             , [me](PersonPtr child){
        return me && child && (child == me);
             });
    }
  }
  return false;
```


© 2022 Peter Sommerlad

Speaker notes

```
}

void testIfChildOfWorks(){
  auto anakin = std::make_shared<Person>("darth vader");
  auto luke = anakin->spawn("luke skywalker");
  auto leia = anakin->spawn("leia organa");
  ASSERT(luke->isChildOf(anakin));
  ASSERT(leia->isChildOf(anakin));
  leia->parent.reset();
  ASSERT(!leia->isChildOf(anakin));
  ASSERT(luke->isChildOf(anakin));
  anakin.reset();
  ASSERT(!leia->isChildOf(anakin));
```

© 2022 Peter Sommerlad

Heap Memory Take Aways 🍔🍟

- prefer `std::unique_ptr` over `std::shared_ptr`, never use `T*` for owning heap memory
- prefer `std::vector`/`std::string` to heap-allocated arrays
- use a library for object graphs
  - if DIY, avoid circular dependencies from `shared_ptr`
    - use `std::weak_ptr` to break such cycles
- `std::shared_ptr` copying can be slow

🚫 🚫

~~`new T{}`~~ ~~`delete p`~~

© 2022 Peter Sommerlad

Speaker notes

Do not use `NULL` or `0` for pointers that are invalid, but use **`nullptr`**

© 2022 Peter Sommerlad

Exercise 7

exercises/exercise07.md

© 2022 Peter Sommerlad

Speaker notes

https://github.com/PeterSommerlad/CPPCourseAdvanced/blob/main/exercises/exercise07.md

© 2022 Peter Sommerlad

Strong Types

Because Tony van Eerd says so:

Tony Van Eerd
@tvaneerd

Type-errors are less likely than many other types of errors.

This is why strong types are worthwhile.

21:36 · 07.01.21 · Twitter Web App

© 2022 Peter Sommerlad

Speaker notes

Tony van Eerd is a C++ friend of mine and I recommend that you watch his conference talks they are very entertaining.

Types were invented to detect wrong and inappropriate computations in programs, i.e., incrementing a boolean value instead of an integer.

With types established also compiler-induced automatic type conversion became possible.

Since C++ ist statically typed, type errors are detectable at compile time and thus are easier to fix than with dynamically typed languages, where type errors often only can be detected at run-time through elaborative automated tests. In addition the static type checking of C++ allows for features such as overloading, selecting the appropriate operation or function for a given type, leading to its unique compile-time polymorphism and enabling the code abstraction through templates without requiring run-time overhead for generic code.

A video of this part from C++Now 2021 is available at https://youtu.be/ABkxMSbejZI

© 2022 Peter Sommerlad

Motivation

- Order of Argument Bug Prevention
- Communicate and Check Semantics of Values
- Limit Operations to Useful subset

© 2022 Peter Sommerlad

Speaker notes

Often, strong, user-defined types for values are motivated with functions taking multiple arguments of the same type but with different meanings. But we also look at two more important reasons for user-defined value types: communicate semantics and limit the amount of operations to a useful subset.

© 2022 Peter Sommerlad

Order of Arguments

```
1  double consumption(double litergas, double kmdriven){
2    return litergas/(kmdriven/100);
3  }
4
5  void demonstrateStrongTypeProblem() {
6    ASSERT_EQUAL(8., consumption(40,500));
7    ASSERT_EQUAL(8., consumption(500,40));
8    // which one is correct?
9  }
```


© 2022 Peter Sommerlad

Speaker notes

Having a function with multiple parameters of the same or compatible types are prone to be called with the wrong order of arguments. In our example, computing the consumption of a car from the amount of gas used vs the kilometers driven it is hard to judge, which test case is actually the intended one, unless we look at the function's implementation. Just using the parameter name can suffice in environments where human code reviews happen for such things, but still any later change can still easily mess up the arguments.

© 2022 Peter Sommerlad

Whole Value Pattern

Ward Cunningham - CHECKS Pattern Language http://c2.com/ppr/checks.html

Because bits, strings and numbers can be used to ***represent almost anything****, any one in isolation* ***means almost nothing****.*

© 2022 Peter Sommerlad

Speaker notes

While the original Whole Value pattern text reflects popular object-oriented programming languages at the time. The underlying paradigm of user-defined types with corresponding operations is a major contribution of OOP and is often taken as a given in programming languages today. C++ has the unique benefit, that such user-defined types can be implemented without run-time overhead (modulo some bad ABI decisions for passing class objects made in the 1990s by some vendors).

© 2022 Peter Sommerlad

Whole Value Pattern

Ward Cunningham - CHECKS Pattern Language http://c2.com/ppr/checks.html

Because bits, strings and numbers can be used to ***represent almost anything****, any one in isolation* ***means almost nothing****.*

Instead of using built-in types for domain values, provide **value types** wrapping them.

© 2022 Peter Sommerlad

Whole Value Pattern: How to

Define value types and use these as **parameters**.

© 2022 Peter Sommerlad

Speaker notes

We are used to use the built-in primitive types in many examples, even so as the pattern says, if a type can represent almost anything, it means almost nothing. Therefore, define and use your own types for your domain values. And in this talk, I'll show you, with how little (own) code such is possible without run-time or space overhead.

© 2022 Peter Sommerlad

Whole Value Pattern: How to

Define value types and use these as **parameters**.

- Provide only useful operations and functions.

© 2022 Peter Sommerlad

Whole Value Pattern: How to

Define value types and use these as **parameters**.

- Provide only useful operations and functions.
- Include formatters for (input and) output.

206

© 2022 Peter Sommerlad

Whole Value Pattern: How to

Define value types and use these as **parameters**.

- Provide only useful operations and functions.
- Include formatters for (input and) output.
- Do not use string or numeric representations of the same information.

© 2022 Peter Sommerlad

Only Useful Operations ?

© 2022 Peter Sommerlad

Only Useful Operations (1)

```
ASSERT_EQUAL(42.0, 7. * ((((true << 3) * 3) % 7) << true));
```


© 2022 Peter Sommerlad

Speaker notes

Unfortunately, C++ suffers from its C legacy with a plethora of operations available for built-in types and worse, with implicit conversions that too easily allows one to mix seemingly inappropriate types in a single expression. For example, if a variable is used to represent a hardware register with flags, the bit-operations might be useful for this variable's type, but it is a strange thing to multiply such a variable with PI(π), a floating point number. So restricting the possible operations while still allow for intuitive use of operators is a useful property of user-defined value types further limiting the chance for errors, that might otherwise go undetected at compile time.

© 2022 Peter Sommerlad

Only Useful Operations (1)

```
ASSERT_EQUAL(42.0, 7. * ((((true << 3) * 3) % 7) << true));
```

C++ has too many operations for its built-in types

- unary: `+ - ++ -- ~ ! * &`
- binary: `+ - * / % && ||`
  `& | ^ << >> == != < <= > >= <=> []`

and allows to mix different types with them through **integral promotion** and **implicit arithmetic conversions**.

© 2022 Peter Sommerlad

Only Useful Operations (2)

```
auto four = 🍎🍎 + 🍎🍎;   // ✅ addition
auto what = 🍎🍎 + 🍐🍐🍐; // ❌
auto huh  = 🍎🍎 * 🍎🍎🍎; // ❌
auto six  = 2 * 🍎🍎🍎;   // ✅ skalar multiplication
```

linear operations are common: `+ -` with scalar: `* /`

© 2022 Peter Sommerlad

Speaker notes

It makes sense to add two apples and two apples to obtain four apples. However, adding apples and pears might be ok for a fruit juice but not in general, because they describe semantically different things. So it would be nice if the plus operator would not work in that case. Also multiplying two apples by three apples is hardly useful. While multiplying 2 meters by 3 meters might be useful, the resulting value must also take into account a change in meaning to be 6 square meters. Hoever, doubling or halfing the amount of apples is a useful operation in addition to addition and subtraction. Such skalar multiplication and addition/substraction abilities, while staying the domain of the type is very common, therefore, it may be nice to allow such grouping of operations, if a more limited set is not useful.

Also, because we want to test our code and because the strong types we are talking about here, should be comparable, we see now.

© 2022 Peter Sommerlad

Simplest Strong Typing

© 2022 Peter Sommerlad

Simplest Strong Typing (1)

`struct`

© 2022 Peter Sommerlad

Speaker notes

Just using a struct to wrap a single member of a built-in type, such as double makes a distinct type for our domain value we want to represent.

You might ask, why not just use the long-term practice of type aliases/`typedef` names. Those do not introduce new types and go no further than parameter names to distinguish your domain types.

A big advantage of this simples strong type approach is that it is also feasible in C to prevent wrongly typed/ordered function arguments.

© 2022 Peter Sommerlad

Simplest Strong Typing (2)

Just use a **`struct`** - works even in C!

```
struct literGas{
    double value;
};
struct kmDriven{
    double value;
};
double consumption(literGas liter, kmDriven km){
    return liter.value/(km.value/100);
}
```

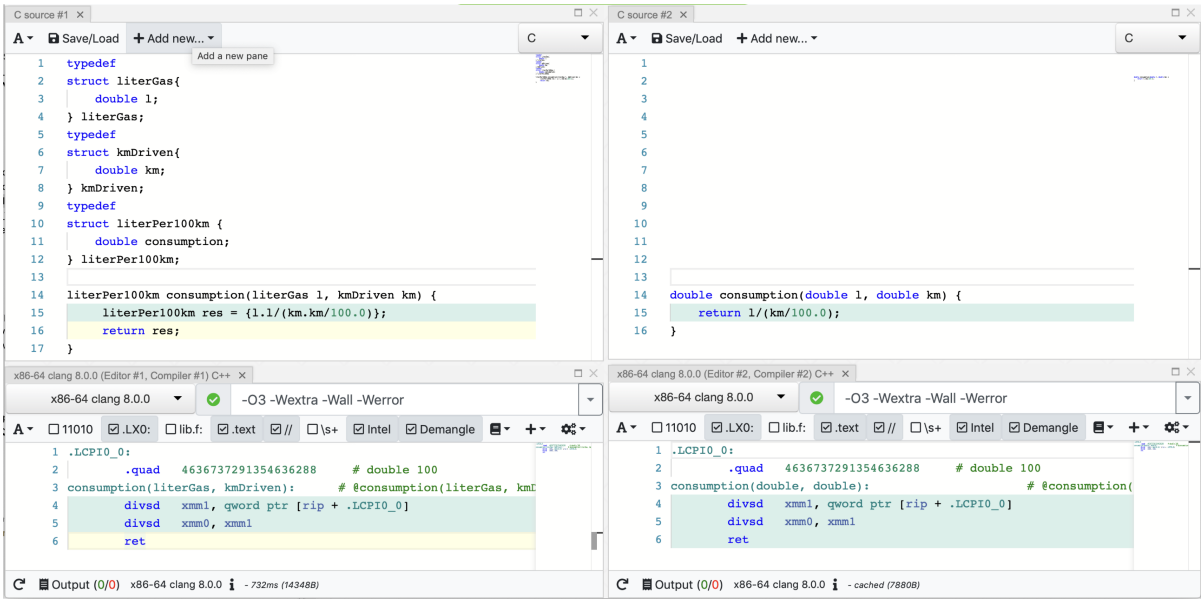

© 2022 Peter Sommerlad

Speaker notes

By just wrapping the values for the domain types in a struct allows us to eliminate the possible confusion (almost). At least it also allows us to overload both sequences thus making the code implicitly correct. However, the implementation got a bit more complex, because we now have to obtain the double value from the wrapper type. We will see later how this can be circumvented by mixing-in operators as hidden friends.

© 2022 Peter Sommerlad

Simplest strong types are efficient



https://godbolt.org/z/d-H6dm

© 2022 Peter Sommerlad

Speaker notes

unfortunately some ABIs (application binary interface) aka C++ Calling Conventions are not as efficient for class types as they are for built-in types of the same size

© 2022 Peter Sommerlad

Simplest Strong Typing (3)

Solution is not perfect, yet.

```
void demonstrateStrongTypeProblem() {
    literGas consumed{40};
    kmDriven distance{500};
    ASSERT_EQUAL(8., consumption(consumed,distance));
    ASSERT_EQUAL(8., consumption({500},{40}));
    // still not perfect, but needs at least {}
}
```


© 2022 Peter Sommerlad

Speaker notes

However, if we do not provide an overload, then the implicit conversion from an initializer list with a single element, still allows to write hard to judge function calls. But at least, these calls are not taking arbitrary numbers anymore. A way to circumvent this ability is to provide the second overload (and make it deleted), to avoid having calls the wrong way around or using the implicit conversion. Those would fail due to ambiguity.

© 2022 Peter Sommerlad

Simplest Strong Typing (4)

consumption() should return also a strong type!

```
literper100km consumption(literGas liter, kmDriven km){
  return {liter.value/(km.value/100)};
  // braces needed (aggregate initialization)
}
void demonstrateStrongTypeProblem() {
  literGas consumed{40};
  kmDriven distance{500};
  ASSERT_EQUAL(literper100km{8}, consumption(consumed,distance));
  // error: no match for 'operator=='
}
```


© 2022 Peter Sommerlad

Speaker notes

Unfortunately, for we do not get the equality comparison for our simplest strong types. However, C++20 provides a way to get comparison operators generated by the compiler with `=default`.

© 2022 Peter Sommerlad

Almost Simplest ST (1)

C++20: default comparison

```
struct literper100km {
  double value;
  constexpr bool
  operator==(literper100km const&) const noexcept = default;
  // C++20 defaulted equality
};
```

equality (`==`) also implies inequality (`!=`)

must return `bool`, param as **`const&`** and be **`const`**

but duplication for every value class

© 2022 Peter Sommerlad

Speaker notes

However, C++20 provides a way to get comparison operators generated by the compiler with **`=default`**.
Unfortunately, we must add the defaulted definition to each of our strong type structs. Another benefit of C++20 is, that a defined equality comparison directly implies availability of a defined inequality comparison (**`operator!=`**).

© 2022 Peter Sommerlad

Almost Simplest ST (2)

C++20: 🛸 <=> all comparisons as defaults

```
struct literper100km {
  double value;
  constexpr auto
  operator<=>(literper100km const &) const noexcept = default;
  // C++20 defaulted 3way comparison
};
//...
  ASSERT(consumption(literGas{40}, kmDriven{500})
         < consumption(literGas{9}, kmDriven{110}));
```

defaulted spaceship also provides equality

< <= > >= == !=

© 2022 Peter Sommerlad

Speaker notes

Instead of equality comparison we can also default the 3way comparison operator (aka spaceship operator). If the spaceship operator is defined as `=default` then all 6 possible comparison operators are defined.

Q: Should we define comparison operators across different types, e.g., literGas and kmDriven ?

A: No, in general comparing apples and pears is not useful, because they represent different things.

© 2022 Peter Sommerlad

operator<=> = default

```
struct literGas {
  double value;
  constexpr auto
  operator<=>(literGas const &) const noexcept = default;
};
struct kmDriven {
  double value;
  constexpr auto
  operator<=>(kmDriven const &) const noexcept = default;
};
struct literper100km {
  double value;
  constexpr auto
  operator<=>(literper100km const &) const noexcept = default;
};
```

still duplication for every value class

© 2022 Peter Sommerlad

Speaker notes

Similar to default definitions of equality, the spaceship operator definintion has to be repeated for each of our strong type classes.

© 2022 Peter Sommerlad

Formatter for Output

```
std::ostream& operator<<(std::ostream &out, literGas const &val) {
  return out << val.value;
}
std::ostream& operator<<(std::ostream &out, kmDriven const &val) {
  return out << val.value;
}
std::ostream& operator<<(std::ostream &out, literper100km const &val) {
  return out << val.value;
}
```

could use unit as decoration

still duplication needed

© 2022 Peter Sommerlad

Speaker notes

For testing and other purposes it would be nice to be able to output the value of our strong type. This allows to understand a test failure for example, because we can inspect the mismatching value directly from the test case output.

But providing an overload for each strong type class in the same namespace the the class is tedious. A benefit of this approach is that we can provide a unit indicator in the output, i.e. 'l' for literGas 'km' for kmDriven, and 'l/100km' for our consumption computation result.

© 2022 Peter Sommerlad

Eliminate Duplication

© 2022 Peter Sommerlad

Speaker notes

Too much boilerplate code is tedious and often error prone due to code cloning and modification.

Let us look at approaches to eliminate such boilerplate code.

© 2022 Peter Sommerlad

Function Templates

```
// full generic
template<typename T>
std::ostream& operator<<(std::ostream &out, T const &val) {
  return out << val.value;
}
```

- must be in namespace of value classes to be picked by ADL
- may be picked up by too many classes (concepts can help)
- assumes specific public member variable

© 2022 Peter Sommerlad

Speaker notes

C++ templates are the key to avoid copy-paste for code that is only different with respect to the types used. We also benefit from C++'s template argument deduction which allows us to just call a function template without the need to specify its template arguments.

© 2022 Peter Sommerlad

CRTP - Curiously Recurring Template Parameter Pattern

```
template<typename derived>//
struct base {
// provide something with derived
};
struct X : base<X>{};
struct Y : base<Y>{};
```

Often used to mix in operators for derived into multiple derived (as Hidden Friends)

© 2022 Peter Sommerlad

Speaker notes

The “Hidden Friends” pattern for operator overloads is also known as the “Barton-Nackman trick”. It can be applied directly within a class, such as putting our `friend operator<<` from above within the class itself.

But for generic inclusion of such hidden-friend operators we apply the CRTP pattern and define the friend operator overloads in a future base class. Since this class does not define any data members it is an “empty base class”. Such empty base classes do not occupy extra space in the object of a derived class.

From C++17 on such empty base classes allow the derived class still to be considered an aggregate. The latter has efficiency benefits for the generated code (trivial copying, potential to pass in registers).

© 2022 Peter Sommerlad

CRTP for output (1)

```
template<typename U>
struct Out {
  friend std::ostream&
  operator<<(std::ostream &out, U const &r) {
    // operator overload as hidden friend
    return out << r.value; // assumes member .value
  }
};
struct literper100km: Out<literper100km> { // CRTP Pattern
double value;
};
```


© 2022 Peter Sommerlad

Speaker notes

Here we see the CRTP pattern applied to provide an output operator for our simple strong type, by inheriting from it.

In this implemenation the CRTP class template assumes all derived classes will provide a data member that has the name `value` and who's type has a corresponding output operator, such as all the built-in types have.

This generic programming by having a member name convention can be used since C++98 and is the key to making templates work. However, it also limits the flexibility of the programmer and can result in interesting template instantiation error messages, when a typo results in a name mismatch.

C++20 concepts in principle allow a way out, by requiring such properties from their template parameters. However, in the case of the CRTP pattern the template parameter cannot be checked when the class template is instantiated, because in that situation the template argument (`literper100km`) is still incomplete and thus a requires clause cannot check its properties.

© 2022 Peter Sommerlad

CRTP for output (2)

Apply structured binding to allow arbitrary member names.

```
template<typename U>
struct Out {
  friend std::ostream&
  operator<<(std::ostream &out, U const &r) {
    auto const& [v] = r; // structured binding
    return out << v;
  }
};
```

Derived class (U) must have a single public member variable

© 2022 Peter Sommerlad

Speaker notes

If we do not want to commit on a naming scheme for our strong type data member, we can assume that it has a single public member variable. This allows us to use C++17 structured binding to access this sole data member in a generic way for all our operator mix-ins without the need for a common naming convention.

© 2022 Peter Sommerlad

Intermezzo: Structured Binding

```
auto [x,y] = f_returningstruct();
```

- On the fly decompose
  - `struct`, `std::tuple<>`, `std::pair<>`
- Number of names in `[ ]` same as data members or elements
- usually `auto` or `auto const &`
- `auto &` only if rhs-function returns lvalue-reference
- *not (yet) possible:* `auto [x...] = f()`

© 2022 Peter Sommerlad

Speaker notes

Structured bindings are usable with tuples and pair, but the most benefit we get from applying it to classes with public data members, because a named type and data member can communicate its purpose better than abstract names like `first` and `second` or `std::get<0>(tuple)`.

Unfortunately, we must know the “arity” of the type (number of non-static data members) and all data members must be public to make that work.

© 2022 Peter Sommerlad

CRTP for output (3)

```
template <typename U>
struct Out{
  friend std::ostream&
  operator<<(std::ostream &out, U const &r) {
    if constexpr (detail__::has_prefix<U>{}){
      out << U::prefix;
    }
    auto const &[v]=r;
    out << v;
    if constexpr (detail__::has_suffix<U>{}){
      out << U::suffix;
    }
    return out;
  }
};
```

Employ a naming convention for prefix and suffix

© 2022 Peter Sommerlad

Speaker notes

Using a name convention we can drag in a prefix and/or suffix static member from the strong type to allow for decorating our strong type's output. For example, `literGas` will use a suffix string of "l" on output.

the conditions `has_prefix` and `has_suffix` use the Detection Idiom to determine the availability of the corresponding static class member.

© 2022 Peter Sommerlad

Providing Operators via CRTP

© 2022 Peter Sommerlad

CRTP for simple arithmetic

```
template <typename R>
struct Add {
  friend constexpr R&
  operator+=(R& l, R const &r) noexcept {
    auto &[vl] = l;
    auto const &[vr] = r;
    vl += vr;
    return l;
  }
  friend constexpr R
  operator+(R l, R const &r) noexcept  {
    return l+=r;
  }
}; // similar: Subtraction
```

Arithmetic operators should also allow the corresponding assignment.

© 2022 Peter Sommerlad

Speaker notes

As with boost operators.hpp library binary arithmetic operators can be defined based on their combined assignment operator. But with CRTP we do not even need a subclass define the combined assignment, but can generically implement it, when we can live without information hiding. Remember, we are implementing types that replace built-in type usage and thus encapsulation was never given for them.

Incrementing, decrementing similarly

© 2022 Peter Sommerlad

CRTP for scalar multiplication (1)

```
template <typename R, typename BASE>
struct ScalarMultImpl {
  friend constexpr R& operator*=(R& l, BASE const &r) noexcept {
    auto &[vl]=l;
    vl *= r;
    return l;
  }
  friend constexpr R operator*(R l, BASE const &r) noexcept {
    return l *= r;
  }
  friend constexpr R operator*(BASE const & l, R r) noexcept {
    return r *= l;
  }
};
```


© 2022 Peter Sommerlad

Speaker notes

For scalar multiplication we need a way to specify the underlying member type and then provide multiplication with it.

© 2022 Peter Sommerlad

CRTP for scalar multiplication (2)

```
struct kmDriven: Out<kmDriven>, ScalarMultImpl<kmDriven,double> {
  double km;
};
literper100km consumption(literGas l, kmDriven km) {
  return {{},{},l.liter/(km*0.01).km}; // multiply km by 0.01
}
void demonstrateStrongTypeProblem() {
  literGas l{{}, 40};
  kmDriven km{{}, {}, 500};
  ASSERT_EQUAL(literper100km({}, {}, 8.), consumption(l, km));
}
```

We get rid of the many { } soon.

© 2022 Peter Sommerlad

Speaker notes

However, our mix-in classes would need to provide the factor type but still allow mixing in the derived class. To provide this in a generic way, we use a meta-programming template argument binder and a template alias.

© 2022 Peter Sommerlad

Bit operators

only useful if data member is `unsigned`

```
template <typename R>
struct BitOps {
  friend constexpr R& operator|=(R& l, R const &r) noexcept
  {
    auto &[vl]=l;
    static_assert(std::is_unsigned_v<underlying_value_type<R>>,
        "bitops are only be enabled for unsigned types");
    auto const &[vr] = r;
    vl |= vr;
    return l;
  }
  friend constexpr R operator|(R l, R const &r) noexcept {
    return l|=r;
  }
```

similarly for **&** and **^**

© 2022 Peter Sommerlad

Shift operators

```
template<typename R, typename B=unsigned int>
struct ShiftOps{
friend constexpr R& operator<<=(R& l, B r)  {
  auto &[vl]=l;
  using TO = underlying_value_type<R>;
        static_assert(std::is_unsigned_v<TO>);
  if constexpr (std::is_unsigned_v<B>){
    pssst_assert(r <= std::numeric_limits<TO>::digits);
    vl = static_cast<TO>(vl << r);
  } else {
    auto const &[vr] = r;
    pssst_assert(vr <= std::numeric_limits<TO>::digits);
    vl = static_cast<TO>(vl << vr);
  }
  return l;
} // ...
```


© 2022 Peter Sommerlad

Making Mix-ins work for Simple Strong Types

© 2022 Peter Sommerlad

Problems to overcome

1. Multiple operations require multiple bases
   - `struct lp100km : Cmp<lp100km>, Out<lp100km>`
   - repeating the same CRTP argument
   - more `{}` needed for init

© 2022 Peter Sommerlad

Problems to overcome

1. Multiple operations require multiple bases
   - `struct lp100km : Cmp<lp100km>, Out<lp100km>`
   - repeating the same CRTP argument
   - more `{}` needed for init
2. Empty bases need leading `{}` for init
   - `literper100km({}, {}, 8.1)`
   - trailing empty braces can be elided

© 2022 Peter Sommerlad

Problems to overcome

1. Multiple operations require multiple bases
   - `struct lp100km : Cmp<lp100km>, Out<lp100km>`
   - repeating the same CRTP argument
   - more `{}` needed for init
2. Empty bases need leading `{}` for init
   - `literper100km({}, {}, 8.1)`
   - trailing empty braces can be elided
3. What are useful operator combinations?
   - Bit operations only for unsigned types
   - Linear operations for numbers

© 2022 Peter Sommerlad

Combining mix-in bases

```
// apply multiple operator mix-ins and keep this an aggregate
template <typename U, template <typename ...> class ...BASE>
struct ops : BASE<U>...{};
template <typename V>
using Additive=ops<V,UMinus,Abs,Add,Sub>;
```

- takes a strong type “derived class” U
- takes a list of mix-in base class templates
- instantiates all bases with U

```
struct liter : ops<liter,Additive,Order,Out>{
  double l{};
};
```


© 2022 Peter Sommerlad

Speaker notes

see how we can use template aliases to abbreviate useful combinations the same class template ops also allows to add more to a specific subclass

© 2022 Peter Sommerlad

Operation combinations

```
template <typename V>
using Additive=ops<V,UMinus,Abs,Add,Sub>;
```

The Additive alias combines unary -, addition, subtraction and computing the absolute value.

© 2022 Peter Sommerlad

Speaker notes

The mix-in class template `Abs<V>` is actually wrapping `std::abs()` function for our framework. Used for tests `ASSERT_EQUAL_DELTA`.

© 2022 Peter Sommerlad

Simple Init of SST

1. define an **`explicit`** constructor
   - requires code in SST class
   - member public for structured binding

```
struct literGas : ops<literGas,Additive,Order,Out> {
  constexpr explicit literGas(double lit):l{lit}{};
  double l{};
};
```


© 2022 Peter Sommerlad

Speaker notes

Using an explicit constructor prevents implicit conversions. However, it also means that the type is no longer an aggregate. You might even want to provide a default argument to the constructor parameter to keep the default constructor available. Or you need to resurrect the default constructor by

```
constepxr literGas() = default;
```

© 2022 Peter Sommerlad

Simple Init of SST

2. put data member in first base class object
   - all following sub-objects are empty
   - empty trailing subobjects do not require {}

```
template <typename V, typename TAG>
struct holder {
  static_assert(std::is_object_v<V>,
    "no references or incomplete types");
  using value_type = V;
  V value { };
};
template <typename V, typename TAG, template<typename...>class ...OPS>
struct strong : holder<V,TAG>,ops<TAG,OPS...> {
};
struct literGas : strong<double,literGas,Additive,Order,Out>{
};
```


© 2022 Peter Sommerlad

Speaker notes

Having the sole data member in the first base class means als other empty bases do not require a pair of braces for initialization. The default init of the value will prevent using uninitializes data.

© 2022 Peter Sommerlad

Different Inits for return

```
template <typename T, typename S = underlying_value_type<T>>
constexpr auto
retval(S && x) noexcept(std::is_nothrow_constructible_v<T,S> ) {
    if constexpr (needsbaseinit<T>{})
        return T{{},std::move(x)}; // value in most derived
    else
        return T{std::move(x)}; // value in base or ctor
}
template <typename U>
struct UMinus{
    friend constexpr U
    operator-(U const &r) noexcept(noexcept(-
        std::declval<underlying_value_type<U>>())){
        auto const &[v]=r;
        return retval<U>(-v);
    }
};
```

Useful for unary operators

© 2022 Peter Sommerlad

Speaker notes

A strong type might need to be initialized with a leading brace for its `ops<>` empty base, or without, when its using `strong<>` as its first base class or when it has a constructor. This is needed to compute the return value from the mix-in operator functions. To ease that distinction, we use the retval function template. This is mainly needed in unary operators and for wrapping functions.

© 2022 Peter Sommerlad

Mapping mathematical functions

Abs is a model for further math functions

```
template <typename U>
struct Abs{ // part of Additive for unit tests comparing float types
  friend constexpr U
  abs(U const &r) {
    auto const &[v]=r;
    using std::abs; // allows stacking of strong types
    return retval<U>(abs(v));
  }
};
```

Not all should return the strong type, but those for rounding are (`ceil, floor, trunc, round, nearbyint, rint`)

© 2022 Peter Sommerlad

Speaker notes

Other math functions like exponential and logarithm functions, or trigonometric functions fundamentally change the domain, therefore should return plain type, not the strong type.

© 2022 Peter Sommerlad

Combining mix-in bases (2)

```
template<typename B, template<typename...>class T>
struct bind2{
  template<typename A, typename ...C>
  using apply=T<A,B,C...>;
};
template<typename BASE>
using ScalarMult = bind2<BASE,ScalarMultImpl>;
template<typename TAG,typename BASE,template<typename...>class ...OPS>
using LinearOps = ops<TAG,     ScalarMult<BASE>::template apply,
                      Additive, Order, Value, OPS... >;
template<typename BASE,typename TAG,template<typename...>class ...OPS>
using Linear=strong<BASE, TAG, ScalarMult<BASE>::template apply,
                      Additive, Order, Value, OPS... >;
```

Useful combination Additive and Linear.
But more restrict things can be better.

© 2022 Peter Sommerlad

Speaker notes

// scalar multiplication must know the scalar type

we use template meta-programming to bind a second type argument to return a single-parameter template for scalar multiplication.

can not use underlying_value_type, because V is still incomplete when the base classes are defined

© 2022 Peter Sommerlad

Strong types, simple?

```
struct literGas {
  double value;
};
struct literGas : strong<double, literGas, Additive, Order, Out> {
  constexpr static inline auto suffix = " l";
};
struct literGas : ops<literGas, Additive, Order, Out> {
  constexpr explicit literGas(double lit={}) : l{lit} {}
  double l;
  constexpr static inline auto suffix = " l";
};
```


© 2022 Peter Sommerlad

Summary: SST Design Options

for mix-in hidden friends with CRTP bases

Version	possible design
C++11	member name convention (can be private)
C++17	public member with structured binding & initial base class with member
C++20	plus operator<=>

© 2022 Peter Sommerlad

PSsst design

- just one data member
  - simply wrap primitive types
  - `is_trivially_copyable_v<strong<T>>`
- Easy mix-in for operators
  - while allowing DIY
- Employs empty-base optimization
- pretends that aggregate is good enough
  - no need for constructors
- as simple as possible for users

caveat do not **`delete`** *via base pointer*

© 2022 Peter Sommerlad

Speaker notes

adding a base class delete protection via a protected base class destructor eliminates the aggregate property of the types.

As a follow up, one needs to define constructors for the strong type, eliminating the simplicity principle.

Guideline: do not allocate PSsst strong type classes on the heap, or at least, do not convert a unique_ptr to a unique_ptr.

© 2022 Peter Sommerlad

STL suffers from primitive obsession

C++ lib uses built-ins where they do not fit well

- `size_t`, `size_type` --> count elements
  - natural numbers including 0
  - **absolute value**
- `ptrdiff_t`, `difference_type` --> distance in sequences, difference between counts!
  - **relative value**

*relative values form **affine space***
*absolute values form **vector space** with an origin*

© 2022 Peter Sommerlad

Standard library lacking strong types

```
size_type __n = std::distance(__first, __last); // implicit conversion to unsigned
if (capacity() - size() >= __n) // aha to avoid warning in comparison
{
   std::copy_backward(__position, end(),
   this->_M_impl._M_finish
      + difference_type(__n)); // cast to the real thing again
   std::copy(__first, __last, __position);
   this->_M_impl._M_finish += difference_type(__n); // and cast again!
```

- warnings often silenced with arbitrary casts

every cast is a indicator for a design improvement waiting

© 2022 Peter Sommerlad

Speaker notes

This led to a whole bunch of discussions around if size should be signed. That is doctoring around the symptom, but not curing the disease.

© 2022 Peter Sommerlad

Absolute-Relative companion types

`<chrono>` is a good example

- `time_point` absolute (vector space)
- `durcation` relative (affine space)
- `tp1 - tp2` --> `duration`
- `tp + d` --> `time_point`
- `d1 + d2` --> `duration`
- `tp1 + tp2` --> **nonsense**

position vs. direction in linear spaces

- often identical representation: bad!
- location vs. displacement
- unit frameworks must make this distinction

© 2022 Peter Sommerlad

Speaker notes

for unit implementors it is easy to neglect the distinction between vector space and affine space. Especially in dimensional analysis it gets lost and then problems occur.

graphics/games software uses vec3d/vec3 or quaternion data structures for location and displacement and thus easily error prone situations occur.

© 2022 Peter Sommerlad

Example using vector/affine spaces

```
struct degrees : Linear<double, Out>{};
struct Kelvin  : create_vector_space<Kelvin,degrees>{};
struct CelsiusZero{
    constexpr degrees operator()() const noexcept{
        return degrees{273.15};
    }
};
struct Celsius:create_vector_space<Celsius,degrees,CelsiusZero> {};

void thisIsADegreesTest() {
    degrees hotter{20};
    Celsius spring{15};
    ASSERT_EQUAL(Celsius{35},spring+hotter);
}
```


© 2022 Peter Sommerlad

Conversion between vector spaces

vector spaces with the same affine space can convert automatically.

```
template<typename TO, typename FROM>
constexpr TO convertTo(FROM from) noexcept{
  static_assert(std::is_same_v<
      typename FROM::affine_space
      ,typename TO::affine_space>);
  return detail_::retval<TO>((from.value-(value(TO::origin())-
        value(FROM::origin()))));
}

void testCelsiusFromKelvin(){
    Kelvin zero{273.15};
    zero += degrees{20};
    ASSERT_EQUAL(Celsius{20},convertTo<Celsius>(zero));
}
```


© 2022 Peter Sommerlad

A glimpse on the "magic"

```
template <typename ME, typename AFFINE, typename ZEROFUNC=default_zero<AFFINE>>
struct create_vector_space : ops<ME,Order, Value, Out>{
  using affine_space=AFFINE;
  using vector_space=ME;
  using value_type=underlying_value_type<affine_space>;
  static inline constexpr auto
  origin() noexcept{ // must be a function and not variable...
    return vector_space{
      detail_::retval<affine_space>(ZEROFUNC{}())};
  }
  // the following two constructors are deliberately implicit:
  constexpr create_vector_space(affine_space v=origin()) noexcept
  :value{v}{}
  constexpr create_vector_space(underlying_value_type<affine_space> v) noexcept
  :value{detail_::retval<affine_space>(v)}{}

  affine_space value;
// operator overloads for linear operations
```


© 2022 Peter Sommerlad

Obtaining the framework

- Peter Sommerlad's Simple Strong Types framework **PSsst** available on github.
- Single header library with a variety of features and freely usable
- Versions for C++17 (main branch) and C++20 (branch).
- A version for earlier C++ is in alpha using `.value` as the data member name convention.

© 2022 Peter Sommerlad

PSsst:

https://github.com/PeterSommerlad/PSsst

© 2022 Peter Sommerlad

PSsst: Outlook? and Take aways 🍔 🍟

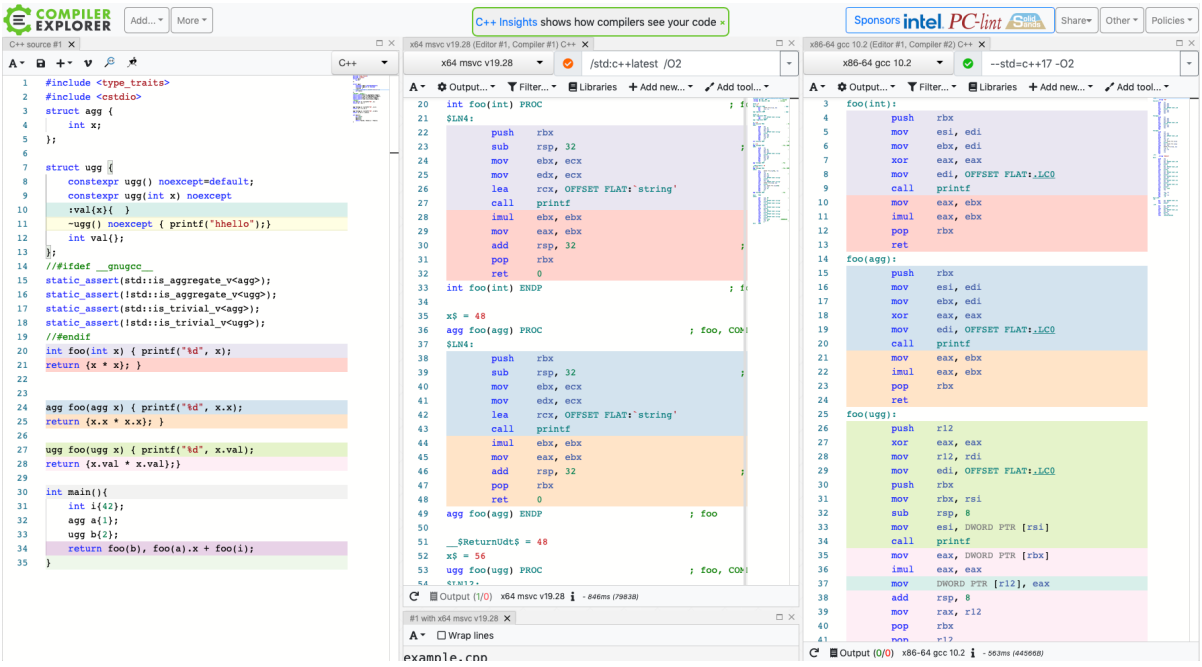
- **More experiments with PSsst are needed**
  - integrate Safe Numerics or similar for safety?
  - MSVC seems to be bad in optimizing aggregates
- **Learn to embrace your language's type system!**
  - Every type cast is a potential design issue!
  - Apply the Whole Value Pattern
  - Check out Ward Cunningham's CHECKS pattern language for more inspiration
- **Use strong type wrappers with care, but use them or DIY!**
  - regardless who's (Joe Boccara, Björn Fahller, Jonathan Müller, Boost, PSsst)
- **Do not confuse a units-framework with strong type wrapper framework**
  - they serve similar but different purposes, many domains need own units!

© 2022 Peter Sommerlad

Extras

© 2022 Peter Sommerlad

Why aggregate?



https://godbolt.org/z/xWfnM5

© 2022 Peter Sommerlad

© 2022 Peter Sommerlad

Speaker notes

C++17 extended the definition of an aggregate to include public inheritance (without special member functions defined). C++20 refined it again to remove unexpected behavior. aggregates do not require a constructor and still can be initialized. Older style was POD, but that didn't allow for member functions.

https://godbolt.org/z/xWfnM5

© 2022 Peter Sommerlad

Detection Idiom vs **concept**

```
template<typename U, typename = std::void_t<>>
struct has_prefix
: std::false_type {
};
template<typename U>
struct has_prefix<U, std::void_t<decltype(U::prefix)>>
: std::true_type {
};
```

checks for possibility of expression, similar to concept

```
template<typename U>
concept has_suffix = requires (U u) { U::suffix; };
```


© 2022 Peter Sommerlad

Speaker notes

Many things that use C++20 concepts can be implemented in earlier C++ with the detection idiom, just more cumbersome.

For some concepts, employing the detection idiom/SFINAE class types, is still useful implementation technique.

© 2022 Peter Sommerlad

Exercise 8

- exercises/exercise08.md

© 2022 Peter Sommerlad

Speaker notes

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

© 2022 Peter Sommerlad

Done...

Feel free to contact me @PeterSommerlad or peter.cpp@sommerlad.ch in case of further questions

© 2022 Peter Sommerlad

Speaker notes

stay tuned for C++Expert if you really want to know how to create modern good C++ libraries.

Unfortunately, even 12 days of intensive training with a lot of exercises in between can not cover all of the nasty details one can encounter.

© 2022 Peter Sommerlad