C++ Expert

C++ Expert

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad@mastodon.social (🦣)

Slides:

https://github.com/PeterSommerlad/CPPCourseExpert/

My philosophy

Less Code

=

More Software

Speaker notes

I borrowed this philosophy from Kevlin Henney.

What is in C++

Tony Van Eerd
@tvaneerd

Replying to @TartanLlama @pati_gallardo and @Cor3ntin

C++ is two languages: the library language, and the application language.
You typically only need one library developer per team, the rest can mostly stick to business logic.
The fact that they all use approx the same programming language is just a bonus for moments of crossover.

8:17 PM · Apr 2, 2022 · Twitter for Android

Speaker notes

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

In C++ Advanced 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).

This course 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. However, I will refrain from using “bad stuff”, whenever there is a cleaner modern solution available. See the notes like here for further old style variations that you might encounter.

© 2024 Peter Sommerlad

In this course

- Modern C++17 and parts of C++20
- IDE Cevelop
- C++ Unit Testing Easier library (CUTE)

6

Speaker notes

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

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

Not in this course

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

Speaker notes

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

We will not look at all features of C++20 and only some of the limitations of previous C++ language standards.

© 2024 Peter Sommerlad

C++ Resources

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

Speaker notes

complete link texts, PDF generation via browser causes hyperlinks to vanish from slide part, but should stick in the notes part:

- 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/CPPCourseExpert

C++ Genealogy

C++98
initial standardized version

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

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

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

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

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

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

Speaker notes

The ISO standardization process uses a three year release cycle since 2011. 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. The ranges library similarly is not "complete" for C++20 in all compilers, so I won't cover it here.

© 2024 Peter Sommerlad

Quick Overview of Modern C++

We can switch to C++ Introduction/Advanced slides to go into details

Speaker notes

Please do not hesitate to ask, if I am talking about something that you didn’t yet have experienced.

Value Types

“When in doubt - do as the `int`s do” – Scott Meyers

- Regular Types:
  - Default Constructible
  - Copyable
  - (Equality Comparable)

Refrain from implicit conversions!

Speaker notes

for your own notes

Algorithms over Loops

employ the “canned” loops in the standard library

- `#include<algorithm>`
- `#include<numeric>`
- `#include<iterator>`

Algorithms come with parallelized overloads

13

Speaker notes

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

The parallelizable algorithms come with an overload that takes an `std::execution_policy` parameter.

- `seq` : sequential (default) iteration
- `par` : allows parallel (multiple-threads) execution
- `par_unseq` : allows vectorization and multiple threads
- `unseq` : allows vectorization on a single thread

Code employing the non-sequential versions of the algorithms is required to not incur data races or deadlocks. Usually the input ranges have to be random access to allow automatic parallelization.

Vectorization usually is valid, when elements touched by a single vector-instruction are contiguous in memory, such as provided by `std::vector` or `std::array`. This sometimes leads to the question if one keeps such data as a vector of structs vs. a struct of vector. The latter keeps data from a single dimensions contiguously and associates data from different dimensions through a common index.

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
5. pass by forwarding reference `(T&&)` -> perfect forwarding

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

Speaker notes

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

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

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

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

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

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

© 2024 Peter Sommerlad

What Classes We Design and How

a mental model for class design

C++ object Roles:

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

C++ specific:

- **Manager** - clean up

Speaker notes

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.

How to implement Manager types is one of the main topics this week.

© 2024 Peter Sommerlad

Manager Classes 👔 for a Resource

*Manage a **single** resource*

- **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, Move-operation for optimization
  - Resource can be (expensively?) duplicated

Speaker notes

Have a non-empty destructor body, e.g., for cleaning up!

Scoped Manager } 👔📦

```cpp
struct Scoped {
  [[nodiscard]] Scoped(); // acquire resource
  ~Scoped();// release resource
  Scoped& operator=(Scoped &&other) = delete;
private:
  Resource resource; // only one!
};
```

Constructor usually has parameters identifying the resource.

DesDeMovA 🙋‍♀️
Rule of if
Destructor defined
Deleted
Move **A**ssigment

Speaker notes

A scoped manager usually does not have a default constructor, but one that takes an identification for the resource to allocate.

Destructor definition has a non-empty body.

If acquisition can fail **AND** exceptions are disabled: make constructor private and have a factory function that returns an `optional<Scoped>` or `variant<Scoped,Error>`.

© 2024 Peter Sommerlad

Unique Manager 🏝️

```cpp
class Unique {
    std::optional<Resource> resource;
    void release() noexcept;
public:
    Unique() = default;
    [[nodiscard]] Unique(Params p); // acquire resource
    ~Unique() noexcept;
    Unique& operator=(Unique &&other) & noexcept;
    Unique(Unique &&other) noexcept;
};
```

`optional<Resource>` provides extra “empty” state for moved-from or default constructed

*New **Rule of Three**, for move-only types*

Speaker notes


```cpp
class Unique {
    std::optional<Resource> resource;
    void release() noexcept;
public:
    Unique() = default;
    [[nodiscard]] Unique(Params p); // acquire resource
    ~Unique() noexcept;
    Unique& operator=(Unique &&other) & noexcept;
    Unique(Unique &&other) noexcept;
};
```


```cpp
Unique::Unique(Unique &&other) noexcept
:resource{std::move(other.resource)}{
    other.resource = std::nullopt; // clear other's optional
}
Unique&
Unique::operator=(Unique &&other) & noexcept {
    if (this != &other) {
        this->release();
        std::swap(this->resource, other.resource);
    }
    return *this;
}
```

```cpp
void Unique::release() noexcept {
    if (resource) {
        // really release resource here
        resource.reset();
    }
}
Unique::~Unique() noexcept {
    this->release();
}
```

© 2024 Peter Sommerlad

Move = Transfer of ownership 🏝️

A📭=std::move(B📫) 👉 A📫, B📭 (actually moved)

```
Unique::Unique(Unique &&other) noexcept
:resource{std::move(other.resource)}{
    other.resource.reset(); // clear RHS optional
}
Unique&
Unique::operator=(Unique &&other) & noexcept {
    if (this != &other) { // self-assignment check
 required
        this->release();
        std::swap(this->resource, other.resource);
    }
    return *this;
}
void Unique::release() noexcept {
    if (resource) { // is optional non-empty
        // really release resource here
        resource.reset(); // AND clear the optional
    }
}
```

```
Unique::~Unique()
 noexcept {
     this-
 >release();
 }
```


Speaker notes

Unique Managers require a deliberate empty “moved-from” state.

using `std::optional` provides the extra “empty” state required for the moved-from state.

If the resources managed is memory, managing that with a `std::unique_ptr` can often employ the default move operations.

New “**Rule of Three** for move-only types”

Instead of assiging `std::nullopt` to `other.resource` in the move constructor, it is also possible to use `other.resources.reset();`, however, some guideline checking tools might highlight that you use a variable in its moved-from state other than destroying it or assigning to it. Using swap as shown in the move assignment operator can be suboptimal, because it might cause two copies of the resources handle, which can be relevant, if the size of the resource handle is significant, or if copying it is non-trivial. But that might be a case never happening.

General Manager 💰

```cpp
struct MValue {
  MValue() = default;
  ~MValue();
  MValue(const MValue &other);
  MValue& operator=(const MValue &other) &;
  MValue(MValue &&other) noexcept ; // optional optimization
  MValue& operator=(MValue &&other) & noexcept; // optional optimization
};
```

Move for optimization only through “gut stealing”.

A📪=B📬 👉 A📬, B📪 (actually moved)

Rule of Three(classic) / Rule of Five/Six

Speaker notes

In this course we will exercise implementing General Manager types!

© 2024 Peter Sommerlad

Special Member Functions

Rule of Zero Rulez

Never define a destructor with an empty body

`=default` virtual destructor

3 kinds of Managers 👔

- **Rule of DesDeMovA** least code for non-copyable
- **Rule of Three(new)** for move-only Unique Managers
- **Rule of Three(classic) or Six** for General Managers

Speaker notes

Define a destructor only when you must do it and never define it with just an empty body (use `=default` for virtual destructor in a base class).

have unique and general managers have a default constructor, creating an "empty" managing object that does not own a resource for managing.

Qualify Member Functions

Mark member functions

- with side effects using **&** ref-qualifier
- without side effects on (***this**) using `const`

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

It is a legacy language design error allowing unqualified (non-const) member functions on temporary objects

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.

Returning a reference to the guts is a side effect. If leaking references from temporaries should not happen, `=delete` the `&&` overlaod

Dynamic Memory

- **NO plain pointers** (`T*`), except encapsulated
- 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`~~

Speaker notes

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

We will see, when it might be necessary/desired to use a heap-allocated array.

No manual memory management using `new`, `delete`, `malloc()`, `free()` etc. in modern C++.

© 2024 Peter Sommerlad

Compile-time over Run-time

errors at compile time are cheapest to fix

- **`static_assert`** to ensure assumption
- **`constexpr`**/**`consteval`** functions and variables
- static polymorphism (**`template`**, **`auto`**) over dynamic polymorphism (**`virtual`**)
- distinct (strong) types over primitive types (e.g. `int` `double`)
- minimalist preprocessor use *`#include`* and -guards

Speaker notes

We will look later at how to do compile-time computation

© 2024 Peter Sommerlad

Conscious Error Handling

When a function's contract could be 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**

Or if there cannot be a contract violation:

-1. always succeed

Speaker notes

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

For option 2 consider `std::optional<T>` as a return type.

The functions without UB (0.) and not throwing exceptions (4.) could be marked with `noexcept`. However, doing so can incur an overhead at the call site due to the need to `std::terminate()` in case an exception is thrown nevertheless.

Anything Else?

- Ask me Anything?

26

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Abstraction

Speaker notes

This section gives an overview on the abstraction mechanisms available in C++ without going into details.

It is provided to form a supportive mental model.

If time is brief, we just might skip it.

And later watch Kate Gregory's ACCU 2022 talk on abstraction that is much more elaborate:

https://www.youtube.com/watch?v=Y3wxJD3BpqI

© 2024 Peter Sommerlad

What is Abstraction?

- give a **Name** for *“stuff”*
  - recall/use via **Name**
- hide details behind **Name**
  - encapsulation enables change

29

Speaker notes

Abstraction is a key concept to programming, even when it is often neglected in teaching programming.

© 2024 Peter Sommerlad

Using Abstraction?

- recall via name allows layering
  - details below details
  - abstraction on top of abstraction
- allows to parametrize *"stuff"*
  - recall passes *arguments*
  - *parameters* are substituted with *argument* values
  - more *generic* solution, better reuse

Speaker notes

Once we have "abstracted" a thing by giving its definition a name, we can recall that "thing" without having to repeat its definition.

Abstraction is further the key to allow parameters for "things": placeholders that can be filled in later with arguments.

This is the key mechanism to achieve "more software with less code".

© 2024 Peter Sommerlad

Abstraction Example

```
#include <iostream>
#include <cmath>

double getNumber(std::istream &is){
  double num{};
  is >> num;
  return num;
}

int main(){
  auto const number { getNumber(std::cin) };
  std::cout << "The square root of "
            << number
            << " is " << sqrt(number);
}
```


Speaker notes

this is not a splendid code example. It is just here to demonstrate the parameter/argument mechanism.

https://compiler-explorer.com/z/1PYGvGMj8

© 2024 Peter Sommerlad

Abstraction in C++

what	how
value, expression	(const) variable
computation (sequence)	function
operation	overloading
set of functions	function template
value set + behavior	(class) type
set of types	class template
related stuff	namespace

Speaker notes

this table is just a rough comparison of the C++ features.

Namespaces (such as `std::`) are not really a means of abstraction, but of grouping.

C++20 in addition allows to abstract otherwise implicit requirements on template arguments with concepts.

© 2024 Peter Sommerlad

Parameterization { } ( ) <>

Abstractions can have parameters

- initialization: `var{value}`
- functions: `f(params)`
- templates: `T<tparams>`

arguments: compile time {}()<>, run time (){}

Speaker notes

This is a very rough overview.

© 2024 Peter Sommerlad

C++ Parameterization

We can parameterize several things:

- functions with function parameters
- lambdas with captures
- template with template parameters
  - class templates
  - function templates
  - variable templates
- template parameters:
  - class templates
  - types
  - compile-time values
  - global references
- argument deduction
  - function templates
  - class templates/constructc

Speaker notes

Parameterization is what makes code composable, testable, and reusable.

Relying on global state syntactically, e.g., writing to `std::cout`, makes code untestable and hard to reuse.

Remove unnecessary dependencies to objects/values/types by introducing parameters for them.

C++ strengths

- C++ has powerful abstraction mechanisms
- compile-time type safety
- generates efficient code (no virtual machine)
- cares much about backward compatibility

the last point is responsible for some of C++ weaknesses

Speaker notes

This concludes the more general overview on abstraction and we will look into the technicalities of C++ again.

© 2024 Peter Sommerlad

Exercise 1

exercises/exercise01/

Speaker notes

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

© 2024 Peter Sommerlad

Templates

Compile-time Code Parametrization

- types (and aliases (C++11))
- functions (and lambdas (C++14))
- variables (C++14)
- concepts (C++20)

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Terminology rehash

Template

- definition `template<>`
- instantiation: `std::vector<int>`
- specialization: `template<>...name<>`
- parameter `template<param>`
  - class `template`, `typename`, compile-time value
- argument `std::vector<int>`
- argument deduction

Speaker notes

We can specialize class templates and variable templates to provide definitions for special cases, or to prevent using the template with specific arguments. Every template instantiation is an (often implicit) specialization of a template. While syntactically it is possible to also specialize function templates, this doesn't provide the same special-case selection as with class and variable templates, therefore, we rely on **overloading** for function templates, which provides the special casing and can use either functions or function templates with the same name forming the overload set.

© 2024 Peter Sommerlad

Function Templates

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

https://godbolt.org/z/jr

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

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 allow to parameterize templates with compile-time values, like the number of elements in a `std::array<T,N>`. Originally, provided for such a case, it was discovered during the initial standardization that they allow turing-complete compile-time programming. This unintended consequence of lisp-style programming using templates was later eased by introducing **`constexpr`**(C++11) and **`consteval`** (C++20) keywords and their corresponding uses, which allow most of C++ to be used at compile time with the original syntax.

Compiler explorer: https://compiler-explorer.com/z/jroxGa6a8

Generic Lambdas

```
auto lambdamin = [](auto const &l, auto const &r){ return l<r ? l : r; };
```

*different **`auto`** parameters are independent typename template parameters*

C++20 allows to specify template parameters for a lambda:

```
auto min = []<typename T>(T const &l, T const &r){ return l<r ? l : r; };
```


Speaker notes

In the first case, heterogeneous lambda function argument types are possible. As long as implicit conversions work for the comparison and return, the code compiles:

https://godbolt.org/z/qMed3ef1d

Specifying the template parameter explicitly and thus giving it a name, allows to ensure both arguments are of the same type.

https://godbolt.org/z/ar576Ga1o

However, neither so far prevent calling the lambda with string literals or pointers. This requires C++20 requires.

Constrained Lambdas

- Lambdas can not be specialized or overloaded
- C++ 20 allows to constrain them:

```
auto min = []<typename T>(T const &l, T const &r)
 requires  (std::is_object_v<T>
           && !std::is_pointer_v<T>
           && !std::is_array_v<T>)
  { return l<r ? l : r; };
```


Speaker notes

In addition to the explicit constraints given by the **`requires`** clause, such a lambda still carries the implicit requirements given by its parameter types and implementation body. Here the existence of the comparison operator< and the need for returning is only implicitly specified.

https://godbolt.org/z/c4d6hq7hq

If we want concept-based compile errors, we can chose to ask for the concept in the lambda template parameter

© 2024 Peter Sommerlad

More Constrained Lambda

- replace `typename` with concept

```
#include<concepts>
auto min = []<std::totally_ordered  T>(T const &l, T const &r)->T const &
requires  (std::is_object_v<T>
           && !std::is_pointer_v<T>
           && !std::is_array_v<T>)
{ return l < r ? l : r; };
```

https://godbolt.org/z/4Kas8G49n

Speaker notes

Play with it:

https://godbolt.org/z/4Kas8G49n

unfortunately, we cannot use “totally_ordered” to constrain the individual lambda parameters directly, because that would again allow heterogeneous calls.

Lambda with own concept

constraints can be combined into a concept

```
template<typename T>
concept minpar = std::totally_ordered<T>
            &&  std::is_object_v<T>
            && !std::is_pointer_v<T>
            && !std::is_array_v<T>;

auto min = []<minpar  T>(T const &l, T const &r)->T const &
{ return l < r ? l : r; };
```

https://godbolt.org/z/9E1E1P1M8

more on concepts and SFINAE later...

Speaker notes

For defining concepts it is often the case that one relies on bool variable templates, often from the standard library header `<type_traits>`.

The standard defines some predefined concepts as well, such as the `std::totally_ordered<T>`, most of them in the header `<concepts>`

https://godbolt.org/z/9E1E1P1M8

© 2024 Peter Sommerlad

Class Templates

- generic data structures
- prevent specific template arguments
  `static_assert()` or declare without definition
- more questions?

see C++ Advanced

Speaker notes

see C++ Advanced material

http://localhost:8000/Advanced.html#/class-templates

Parameters

Function parameters

- value
- lvalue-reference
- const lvalue-reference
- rvalue reference

Compile-time template parameters

- types
- constant objects
- functions
- templates
- concepts (for constraints)

Speaker notes

We do not promote pointers as function parameters here, because pointers behave like value parameters, but have the semantics of an optional lvalue-reference. See C++ Advanced course for replacement strategies of pointers (repeated and extended towards the end of this course, time permitting)

© 2024 Peter Sommerlad

Function Parameter Kind

Prefer parameter definitions as follows:

1. **pass by value**
2. *pass by const-reference* ***`const &`***
   -> opt-/pess-imization of 1
3. pass by lvalue-reference **`&`** -> side effect
4. pass by rvalue-reference **`&&`** -> transfer of ownership
5. pass by forwarding reference **`(T&&)`** —> && used in association with template parameter
   -> perfect forwarding

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.

Passing by forwarding reference (deduced r-value reference syntax) is for perfect forwarding.

Move Semantics

1. Transfer of ownership with Unique Managers
2. Copy-optimization through “gut stealing” with General Managers

A📪=B📬 👉 A📬, B📪 (actually moved)

Copy is also a valid move operation

Speaker notes

A type is movable, when it is copyable, but a type can have a dedicated overload of its move operations either, because it is a unique manager for a non-duplicatable resource, or it is a general manager where move operations perform “gut stealing” and thus optimize copy.

1. `std::unique_ptr<T>` - Unique Manager
2. `std::vector<T>` - move optimized General Manager

Some types are so cheap to copy, move support is superfluous (e.g. `int`).

Some types are always expensive to copy and won’t allow move optimization, because they contain all data, e.g., `std::array<T,N>`

© 2024 Peter Sommerlad

Copying Content

A📭=B📫 👉 A📫, B📫 (copy)

```
#include <iostream>
struct CopyableThing {
  CopyableThing() {
    std::cout << "Create Thing\n";
  }
  CopyableThing(CopyableThing const &) {
    std::cout << "Copy Thing\n";
  }
};
CopyableThing create() {
  CopyableThing t{};
  return t;
}
int main() {
  CopyableThing created = create();
}
```

https://godbolt.org/z/EKsoE8nEo

Concept of “copy elision"

50

Speaker notes

https://godbolt.org/z/EKsoE8nEo

Experiment with different language standards with and without -fno-elide-constructors:

- `-O2 -std=c++11 -fno-elide-constructors`: https://godbolt.org/z/EKsoE8nEo
- `-O2 -std=c++17 -fno-elide-constructors`: https://godbolt.org/z/rM9KfMP14
- `-O2 -std-c++17` : https://godbolt.org/z/qbaobqajh

C++17 introduced mandatory copy-elision -> single copy

GCC implements Named-Return-Value-Optimization (NRVO) -> no copy

Moving Content

A📭=B📬 👉 A📬, B📭 (move)

```
#include <iostream>
struct MoveOnlyThing {
  MoveOnlyThing() {
    std::cout << "Create Thing\n";
  }
  MoveOnlyThing(MoveOnlyThing &&) {
    std::cout << "Move Thing\n";
  }
};
MoveOnlyThing create() {
  MoveOnlyThing t{};
  return t;
}
int main() {
  MoveOnlyThing created = create();
}
```

https://godbolt.org/z/abK8q9zaz

Speaker notes

https://godbolt.org/z/abK8q9zaz

Experiment with different language standards with and without -fno-elide-constructors:

- `-O2 -std=c++14 -fno-elide-constructors`: https://godbolt.org/z/64hrPW56b
- `-O2 -std=c++17 -fno-elide-constructors`: https://godbolt.org/z/abK8q9zaz
- `-O2 -std-c++17` :https://godbolt.org/z/59Gd1sEe8

C++17 introduced mandatory copy/move-elision -> single move

GCC implements Named-Return-Value-Optimization (NRVO) -> no move

© 2024 Peter Sommerlad

Where move is relevant?

```
using BigObject = std::array<int, 1'000'000>;
struct ContainerForBigObject {
  ContainerForBigObject()
      : resource{std::make_unique<BigObject>()}
  {}
  ContainerForBigObject(ContainerForBigObject const & other)
      : resource{std::make_unique<BigObject>(*other.resource)} {}
  ContainerForBigObject(ContainerForBigObject && other) noexcept
      : resource{std::move(other.resource)}
  {}
  ContainerForBigObject & operator=(ContainerForBigObject const & other) {
    resource = std::make_unique<BigObject>(*other.resource);
    return *this;
  }
  ContainerForBigObject & operator=(ContainerForBigObject && other) noexcept {
    std::swap(resource, other.resource);
    //resource = std::move(other.resource);// is possible too
    return *this;
  }
private:
  std::unique_ptr<BigObject> resource;
};
```

https://godbolt.org/z/W14Pqzbbh

move operation needed to be explicit here because BigObject doens't have itself an "empty" state, while unique_ptr can have an "empty" state

Speaker notes

Since std::array actually holds its elements within the object, it is big and expensive to copy. Therefore, dynamic allocation makes sense, because it not only prevents overflowing the stack memory, but also allows copy optimization through a move operation.

This is a general manager type for demonstration purposes only, that doesn't require a destructor, but implements copy and move operations. Actually the move operation implementations could be defaulted and will generate identical code, if swap() is replaced with move assignment.

https://godbolt.org/z/W14Pqzbbh

Defaulted move operations:

```
struct ContainerForBigObject {
  ContainerForBigObject()
      : resource{std::make_unique<BigObject>()}
  {}
  ContainerForBigObject(ContainerForBigObject const & other)
      : resource{std::make_unique<BigObject>(*other.resource)} {}
  ContainerForBigObject(ContainerForBigObject && other) noexcept = default;
  ContainerForBigObject & operator=(ContainerForBigObject const & other) {
    resource = std::make_unique<BigObject>(*other.resource);
    return *this;
  }
  ContainerForBigObject & operator=(ContainerForBigObject && other) noexcept = default;
private:
  std::unique_ptr<BigObject> resource;
};
```

https://godbolt.org/z/sesPebnqr

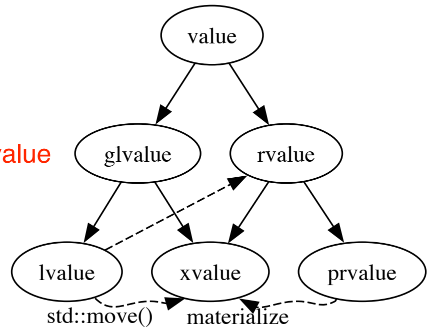
For containers and Manager objects that manage dynamic memory (via `std::unique_ptr` if DIY) for their contents, move-optimization makes sense.

If a class' data members support move optimization, the compiler-provided default move operations (if not suppressed otherwise) readily employ the optimization without the need to implement them oneself for the class.

Excursion: C++ Value Categories

expressions have a type and a value category

generalized l-value



pure r-value

expiring value

value categories

T - (pr)value, **T&** - lvalue, **T&&** - xvalue

The type representation is not exactly correct

int a = 10;
- a: is l-value
- 10: is r-value

53

Speaker notes

original value categories from C contain lvalues and rvalues, where l and r denote the position within an assignment.

lvalues have a space where to store a value. rvalues just are the value that is read/used.

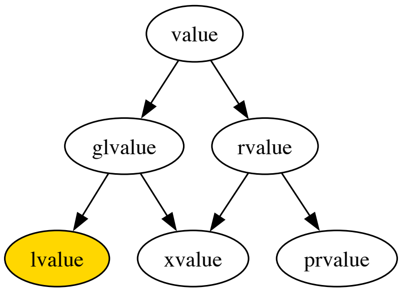
With C++11 introduction of move semantics the value categories have gotten a finder granularity:

- glvalue - represents a location where a value resides (lvalue or xvalue)
- lvalue - represents a location where a value can be stored
- xvalue - represents a value that can be consumed (the moved from value might change on move), eXpiring value
- rvalue - represents a value that is read or used (xvalue or prvalue)
- prvalue - represents a “pure” rvalue that does not need to have a location

Within an expression that just uses the value of a variable, conceptually an implicit lvalue-to-rvalue conversion happens. However, there is no implicit conversion from lvalue to xvalue! A pure rvalue (prvalue), like formed from a literal, e.g., `42`, implicitly converts to an xvalue through a conceptual mechanism called *“materialization”* when passed to a function taking an rvalue-reference parameter. Most of these implicit conversions have any run-time overhead, because they just cover how the compiler treats the expressions’ types.

rvalue-references (**T&&**) actually refer to an **xvalue** and thus should be called *“xvalue-references”* but we cannot change established terminology.

lvalue-references T&



- function parameter for side-effects
- can dangle! 💣
  - return type (must survive call!)
  - member or local variable

never return a local variable by reference

Speaker notes

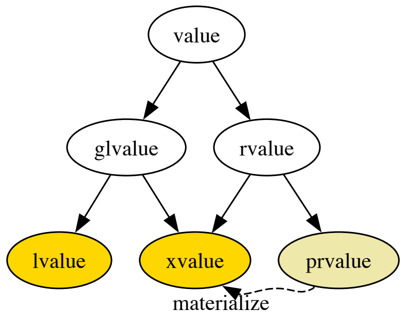
While T `const &` is also an lvalue reference, such const-references when used as parameters can also bind to rvalues (= temporaries).

For local const-references that bind to a (member of a) temporary object that is directly created directly in the binding expression, the lifetime of the temporary is extended until the end of the block. However, slight refactorings or binding to a reference returned from a function won't extend the lifetime and lead to dangling. In addition C++17's mandatory copy elision actually eliminated most cases, where this was used for copy prevention.

I suggest strongly to not rely to temporary lifetime extension by binding it to a local reference. The C++ standard even contains a bug-problem caused by such "optimization" in the specification of the range-for loop statement.

© 2024 Peter Sommerlad

const-lvalue references T `const &`



- parameter for copy-optimization
  - pessimization for “small” types
- can dangle! 💣
  - return type (must survive!)
  - returning a member “ok”

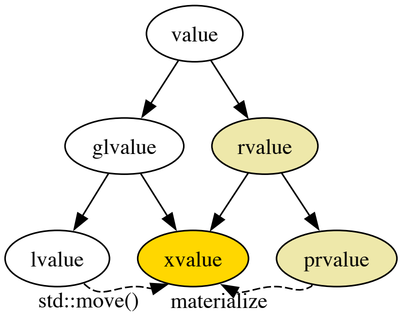
never return a local variable by reference

Speaker notes

Technically a prvalue is “materialized” (dashed arrow) to become a temporary (xvalue), when it has to be bound to a (const-) reference.

Using local variables that are const-references, have an interesting but very error prone effect, that they bind to a temporary and extend its lifetime. Minimal refactorings can break the lifetime extension and thus lead to dangling 💣.

rvalue-references T&&



- function parameter for transfer of ownership
- parameter name is lvalue in function
- *In deduced scope (`auto&&`) does not mean rvalue-reference!*

Speaker notes

Like const-references, local rvalue-references bind to temporary objects and extend their lifetime. However, as with const-references such lifetime extension is error prone and can lead to dangling when code gets refactored. DO NOT USE IT!

Also be aware that an rvalue-reference parameter becomes an lvalue when its name is used within a function.

Having a function with an rvalue-reference return type is most often not very useful (return by value instead if it is a non-local (surviving) variable in the return statement, you might want to use std::move in the return statement to optimize the copy)

Parameter Reference Binding

`struct S`	`f(S)` *value*	`f(S &)` *side effect*	`f(S const &)` *no copy*	`f(S &&)` *transfer of ownership*
`S s{};` `f(s);`	✅	✅ preferred	✅	❌
`S const s{};` `f(s);`	✅	❌	✅	❌
`f(S{});` Since S{} will expire, it's like S has an "empty" state, therefore the signature && works	✅	❌	✅	✅ preferred
`S s{};` `f(std::move(s));`	✅	❌	✅	✅ preferred

- either per value or per reference, combined overloads can cause ambiguities

Speaker notes

Value overload causes ambiguities if also a reference overload exists.

If both const and non-const lvalue reference overload exist, constness of the argument determines which one is selected.

While syntactically possible to from const-rvalue-references, they make semantically no sense, because they are there for "gut stealing"

Overload resolution member functions

`S:: m()` **qualifier:** *usage*	***none*** ***side effect***	`const`	`&` ***side effect***	`const &`	`&&` ***consume***
`S s{};` `s.m();`	✅	✅	✅ preferred	✅	❌
`S const s{};` `s.m();`	❌	✅	❌	✅	❌
`S{}.m();`	✅ 🕳💣	✅	❌	✅	✅ preferred
`S s{};` `std::move(s).m();`	✅ 🕳💣	✅	❌	✅	✅ preferred

side effects on temporaries with unqualified member functions is an unfixable legacy hole 🕳 in the type system

First two columns refers to function m() specifier

Last three columns refers to function parameters

58

Speaker notes

I strongly recommend to qualify member functions that are intended to be called on mutable lvalues!

Unfortunately, within an overload set of member functions, as of up to at least C++20, one needs to either ref-qualify all member functions (new style) or none (old style).

While in theory the syntax for `const &&` exists, it is quite useless in practice.

© 2024 Peter Sommerlad

Simple Type Deduction

- **`auto`**
  - (local) variables (`=` C++11 `{val}` C++14)
  - function return types (C++11(*) C++14)
  - lambda parameters (C++14)
  - function parameters (C++20)
- `template<typename T>void f(T)`
  - function template from call argument
  - class template from constructor argument (C++17)

Deduced type is from value without further qualification.

(*) C++11 **`auto`** return type requires `->` trailing return type

Speaker notes

In C++11 `auto i{42};` would deduce `std::initializer_list<int>` as the type for `i`. This confusing "specification bug" was retroactively fixed so that using curly braces for type deduction from a single value would not trigger deducing `std::initializer_list`. Unfortunately, there are other problems around initializer lists that couldn't be fixed (the priority of selecting an `initializer_list` constructor, even when implicit conversions would be necessary).

The type deduction for **`auto`** is specified in terms of type deduction that happens for **`typename`** function template parameters used as function parameter types.

lvalue-ref Type Deduction

```
auto &
template<typename T> void f(T &)
```

declaration	call	instantiated	deduced T
`int x{1};`	`f(x)`	`f(int &)`	`int`
`int const cx{2};`	`f(cx)`	`f(int const &)`	`int const`
`int const &crx{3};`	`f(crx)`	`f(int const &)`	`int const`

Speaker notes

for your own notes

© 2024 Peter Sommerlad

`const` `&` Type Deduction

```
auto const &
template<typename T> void f(T const &)
```

declaration	call	instantiated	deduced T
`int x{1};`	`f(x)`	`f(int const &)`	`int`
`int const cx{2};`	`f(cx)`	`f(int const &)`	`int`
`int const &crx{3};`	`f(crx)`	`f(int const &)`	`int`

—> not the function
the function variable

Speaker notes

for your own notes

Forwarding Reference Type Deduction

```
auto&&
template<typename T> void f(T&&)
```

declaration	call	instantiated	deduced T
`int x{1};`	`f(x)`	`f(int &)`	`int &`
`int const cx{2};`	`f(cx)`	`f(int const &)`	`int const &`
	`f(42)`	`f(int &&)`	`int` ~~`&&`~~
`int const &crx{3};`	`f(crx)`	`f(int const &)`	`int const &`

`T&&` is a forwarding reference, type deduced keeps "referencyness"

Speaker notes

Note that a regular rvalue-reference function parameter `std::unique_ptr<int> &&` in non-deduced contexts will only bind to xvalues and not to an lvalue.

Such rvalue-reference paramters are only useful to explicitly mark the consumption of a function argument and expect to be called with temporaries. If the underlying type is a move-only type, it is sufficient to use call-by-value for the same effect. So taking ownership via an rvalue-reference parameter is only useful for General Manager types, such as `std::vector<int>` where the "gut stealing" is intentional.

In generic containers insertion member functions often come in overloads for `const&` and `&&` parameters. This allows to optimize for temporaries to be moved into their place in the container and for lvalues to be copied into their place in the container. It is also required to support move-only types in containers, that cannot be simply copied.

© 2024 Peter Sommerlad

`decltype(auto)`

`decltype(expr)` deduces the type of expr

while `auto` deduces always the value type, `auto&&` always deduces a reference, `decltype(auto)` keeps the deduced type's "referenciness"

Speaker notes

The main use of decltype(auto) is for determining the return type of generic functions that might return references.
Here, we also have a lot of potential danger situations that can lead to returning a dangling reference!

© 2024 Peter Sommerlad

decltype(auto) deduction

```
decltype(auto) funcName() {
  int local = 42;
  return local; //decltype(local) =>
        int
}
decltype(auto) funcPrvalue() {
  return 5;  //int
}
```

```
decltype(auto) funcNameRef() {
  static int local = 42;
  int & lref = local;
  return lref; //decltype(lref) => int
        &
}
decltype(auto) funcXvalue() {
  int local = 42;
  return std::move(local); //int && ->
        bad
}
decltype(auto) funcLvalue() {
  int local = 42;
  return (local); //int & -> bad
}
```

- variable or data member name
  - T - type of the expression (retains reference)
  - here lvalue->rvalue implicit conversion happens
- expression of value category prvalue
  - T
- expression of category xvalue
  - T&& - rvalue reference type
- expression of value category lvalue
  - **this includes (name)!**
  - T& lvalue reference

—> pay attention to this: do not put parenthesis around the return value

Speaker notes

type deduction with **decltype(auto)** is tricky, because of the special case of putting parenthesis around a named return value. So as a general rule, never use parentheses around the expression in the return value, since this also prevents application of the optional named-return-value optimization (NRVO).

Useful **decltype()**

In generic inline functions, where returning a–potentially const–reference is useful.

```
template<typename Container, typename Index>
decltype(auto) access(Container & c, Index i) {
  return c[i]; // keep return type of Container::operator[]
}
```

from C++11 on in trailing return type referring parameters

```
template<typename Container, typename Index>
auto access(Container & c, Index i) -> decltype(c[i]) {
  return c[i];
}
```


Speaker notes

In C++11 **`auto`** for function return type, requires to specify the return type after the function parameters using an **`->`** to provide a “trailing return type” specification. The same syntax is available for Lambda expressions, if the deduced type wouldn’t be unique or would be different from the actual type of the return expression.

© 2024 Peter Sommerlad

Generic Wrapping

Move-only types require rvalue-reference or value parameters

```
template<typename T>
struct wrapper{
  wrapper() = default;
  explicit wrapper(T const &x)
  :value{x}{} // copy
  explicit wrapper(T&& x)
  :value{std::move(x)}{}
  T value;
};
```

```
template<typename T>
struct wrapper{
  wrapper() = default;
  explicit wrapper(T x)
  :value{std::move(x)}{}
  T value;
};
```

`std::move(x)`, because named parameter `x` is treated as an lvalue when used.

Speaker notes

play with it: https://godbolt.org/z/EnaEr8s5z

This code is over-simplified to show the different options for constructor parameters. We will extend it further later.

At the moment, we can observe, that the type needs to be *default constructible* and *moveable* (copyable is a special form of movable)

inside `std::move`

- How does `std::move()` move objects?
  - **it doesn't!**
  - it is a `static_cast<T&&>()`
  - `T&` -> `T&&`
- possible implementation:

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<std::remove_reference_t<T>&&>(param);
}
```


Speaker notes

std::remove_reference_t is used to create the T from `T&` lvalue-reference type

decltype(auto) return type retains rvalue-reference as return type.

Some library implementers use the `static_cast<T&&>` directly, because some C++ frontends don't optimize the function call away, as they should IMHO. However, that needs to ensure that the underlying type T is a value type. If it is already a reference, so called reference collapsing kicks in.

Why not just `static_cast<T&&>`?

reference collapsing

- `static_cast<T&&>(x)` might yield an lvalue reference!

Type	Combination	Resulting Type
T	T &	T&
T&	T& &	T&
T&&	T&& &	T&
T	T &&	T&&
T&	T& &&	T&
T&&	T&& &&	T&&

Speaker notes

Reference collapsing only results in an rvalue reference if the rvalue-reference is added to a value type or a type that is already an rvalue reference type. All other cases of reference adding to a type yield an lvalue reference.

© 2024 Peter Sommerlad

Replacing the wrapped object

```
template<typename T>
struct wrapper{
  wrapper() = default;
  explicit wrapper(T
         const &x)
  :value{x}{}
  explicit wrapper(T&&
         x)
  :value{std::move(x)}{}
  T value;
  void replace(T const
         &x) & {
    value = x;
  }
  void replace(T &&x) &
         {
    value =
         std::move(x);
  }
};
```

```
template<typename T>
struct wrapper{
  wrapper() = default;
  explicit wrapper(T x)
  :value{std::move(x)}{}
  T value;
  void replace(T x)& {
    value = std::move(x);
  }
};
```

pass-by-value might need to copy twice: ok for move-optimized or small types

Speaker notes

Assuming the existence of move-assignment, we can also replace the value.

However, neither works for types that cannot be assigned or moved.

© 2024 Peter Sommerlad

Wrapping non-movable?

```
template<typename T>
struct wrapper{
  wrapper() = default;
  explicit wrapper(T const &x)
  :value{x}{} // copy
  explicit wrapper(T&& x)
  :value{std::move(x)}{}
  T value;
};
struct nonmovable{
    nonmovable& operator=(nonmovable&&)=delete; // rule of DesDeMovA
};

wrapper<nonmovable> w{nonmovable{}}; // doesn't compile
```

we need a means to construct an object in place

https://godbolt.org/z/4GKaYEP7z

Speaker notes

https://godbolt.org/z/4GKaYEP7z

We can not use our wrapper with scoped manager types or types that are cannot be moved and thus not copied.

To facilitate such types, we need a means to construct an object in place. This can also be used for objects that are very expensive to copy, because they are large, and that do not benefit from move optimization. The standard library employs such in the `emplace(...)` member function templates of its container class templates.

Wrapping a function

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T arg){
  f(arg); // f(T) or f(T const &)
}
```

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T&& arg){
  f(arg); // f(T), f(T const &), f(T&)
}
```

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T& arg){
  f(arg); // f(T&)
}
```

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T&& arg){
  f(std::move(arg)); // f(T), f(T&&), not f(T&)
}
```

https://godbolt.org/z/Kb6hq4rGM

Perfect Forwarding:

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T&& arg){
  f(std::forward<T>(arg));  —> Allows you to use the correct function overload
}
```


Speaker notes

- Pass by value, only works well for copyable value types.
- Pass by lvalue-reference, only works well with lvalue-reference parameters
- Pass by forwarding-reference, works with value and lvalue-references, but cannot pass-on a move-only type directly, because the parameter itself is an lvalue. ** using `std::move()` is not a solution, because that would prohibit pass-by-lvalue-reference ** using `std::forward<T>()` is the solution

The function parameter is passed by forwarding reference to enable the use of either functions (lvalue), function objects (lvalue), or lambda expressions (xvalue). In theory, it would also benefit from perfect forwarding, but that is not needed here, because plain functions can always be called.

https://godbolt.org/z/Kb6hq4rGM

Perfect Forwarding gotchas

```
struct func{
void operator()(int const & value) {
    std::cout << "call by const&: "
              << value << '\n';
}
void operator()(int &value){
    std::cout << "call by &: "
              << value << '\n';
}
void operator()(int &&value){
    std::cout << "call by &&: "
              << value << '\n';
}
};
```

```
template<typename FUNC, typename T>
auto wrapit(FUNC &&f,T&& arg){
  f(std::forward<T>(arg));
}
int main(){
  int i{41};
  int const ic{44};
  int &&ixref=43;
  func f{};
  wrapit(f,i); // lvalue-ref
  wrapit(f,42); // xvalue-ref
  wrapit(f,static_cast<int const &>(42)); // const lvalue-ref
  wrapit(f,ic); // const lvalue-ref
  wrapit(f,ixref); // lvalue-ref
  wrapit(f,std::move(i)); // xvalue-ref
}
```

https://godbolt.org/z/bEf6KEjd3

Speaker notes

https://godbolt.org/z/bEf6KEjd3

- Passing an rvalue will bind to rvalue-reference
- Passing an lvalue will bind to lvalue-reference
- Passing a const lvalue or a const lvalue reference binds to const-lvalue-reference

Perfect Forwarding for construction

variadic template forwarding parameter…

```
template<typename T>
struct wrapper{
  //construct in-place
  template<typename...PARAMS>
  explicit wrapper(PARAMS &&...args)
  :value(std::forward<PARAMS>
        (args)...){}
  T value;
};
struct nmv_def {
  nmv_def& operator=
        (nmv_def&&)=delete;
};
wrapper<nmv_def> w{};
```

```
struct nmv_2ref {
  nmv_2ref(int &i,
           std::unique_ptr<int>
        p)
  : i{i}, up{std::move(p)}{}
  nmv_2ref& operator=(

        nmv_2ref&&)=delete;
  int &i;
  std::unique_ptr<int> up;
};
int i{42};
wrapper<nmv_2ref> w{i,
  std::make_unique<int>(43)};
```

73

Speaker notes

https://godbolt.org/z/7s57r887n

Using a variadic forwarding parameter pack with std::forward<>() pack expansion is the way for generic forwarding.

© 2024 Peter Sommerlad

Perfect Forwarding Usage

emplace()

- containers use emplace() to prevent additional copies when inserting elements by constructing elements directly through perfect forwarding
  - e.g., `std::vector::emplace_back()`

```
template< class... Args >
reference emplace_back( Args&&... args );
```

- generic wrappers similarly: `std::optional`, `std::any`

Speaker notes

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

https://en.cppreference.com/w/cpp/utility/optional/emplace

https://en.cppreference.com/w/cpp/utility/any/emplace

© 2024 Peter Sommerlad

Parameter summary

- prefer pass-by-value
- use pass-by-lvalue reference for side-effects
- use pass-by-const-reference for copy-optimization
- use pass-by rvalue-reference for taking ownership
  - don't forget `std::move(arg)`
- use forwarding-reference for perfect forwarding
  - don't forget `std::forward<T>(arg)`

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Exercise 2

exercises/exercise02/

Speaker notes

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

© 2024 Peter Sommerlad

Object Lifetime and Raw Memory

In contrast to C, C++ is much more specific about object lifetime.

In contrast to Python, C++ object lifetime is deterministic

Speaker notes

for your own notes

© 2024 Peter Sommerlad

C++ Object Lifetime

- C++ has a deterministic lifetime model
- Lifetime starts, when the first constructor finishes successfully
- Lifetime ends, when its destructor is called
- in principle also for non-class types
- Existence of memory doesn't start lifetime
- re-interpreting bits of an object as another type is invalid in most cases

Accessing an object outside of its lifetime is
Undefined Behavior 💣

Speaker notes

While the concept of type-punning – interpreting bits of one object as an object of another type – is common practice in C, for example with `union`s, there are only very few cases, where doing so is valid in C++. Even with built-in types that share the same layout and range the underlying types are considered distinct by the type system.

This section is dealing with the cases when and how such things are valid in C++. Note, that a successful warning-free compilation not necessarily guarantees that.

Please note, that the C++ standard experts are currently still discussing details of what operations on raw memory are valid or not, based on the standard. (un)fortunately, many such low-level accesses that programmers think should work, work in practice, even when not sanctioned by the standard. However, there are several corner cases, where compilers might take the C++ standard omissions (= undefined behavior) as a source for optimization potential and thus create opportunities for code not behaving in a way the developer expects, especially with optimization turned on.

Dynamic Storage aka Heap Memory

- Use standard library containers, if not sufficient:
- `std::make_unique()` is the modern way for memory allocation

classic:

- `auto ptr = new T()` - heap allocation and construction
- `delete ptr;` - destruction and deallocation

always been obsolete (C):

- `char * ptr = malloc(sizeof(T));`
- `free(ptr);`

Speaker notes

correctly pairing **`new`** and **`delete`** is required to not leak memory, even in case of exceptions and also to not cause undefined behavior, in case of an array allocation, **`delete[]`** needs to be used accordingly for releasing the corresponding memory.

In modern C++, one should exclusively use `std::make_unique<T>()` to allocate objects on the heap, except when one needs shared pointers and then uses `std::make_shared`.

A last expert-level option for specific allocation strategies in containers is to use the `std::allocator_traits` API with dedicated allocators or with the containers from the namespace `std::pmr::` that support polymorphic allocators to provide dedicated memory resource objects derived from `std::pmr::memory_resource`. The latter allows to mix containers with different memory resources, without encoding the allocator in its concrete type, as it is the case with containers from the namespace `std::`

© 2024 Peter Sommerlad

Allocators

Standard containers have an `Allocator` typename template parameter

```
template<class T, class Allocator = std::allocator<T>> class vector;
```

An allocator provides two significant member functions:

```
T* allocate( std::size_t n );
void deallocate( T* p, std::size_t n );
```

the `std::allocator<T>` delegates to plain `::new` and `::delete` and is stateless

Speaker notes

For memory limited devices or specific (concurrent/parallel) architectures it can be useful to use/define dedicated allocators.

However, since the standard container templates are paraemterized by allocator type, mixing default containers with those with a dedicated allocator type is not simple.

The standard library therefore provides containers in namespace std::pmr that allow to mix containers with different allocation strategies, defined by “polymorphic memory resource”. Using those should be clearly motivated by the architecture and make a measurable difference.

Employing Allocators

Indirection through
`std::allocator_traits<Alloc<T>>`

```
[[nodiscard]] static constexpr T* allocate( Alloc& a, size_type n );
template< class T, class... Args >
static void construct( Alloc& a, T* p, Args&&... args );
template< class T >
static void destroy( Alloc& a, T* p );
static void deallocate( Alloc& a, T* p, size_type n );
```

For dedicated allocation strategy consider
`std::pmr::memory_resources` subclasses

pmr: polymophic memory resources

Speaker notes

Types supporting allocators have to follow specific conventions. The type trait `std::uses_allocator<T,Alloc>` can be employed in generic code to determine if allocator support is enabled in type T for a specific allocator type `Alloc`.

https://en.cppreference.com/w/cpp/memory/uses_allocator

Types using allocators, usually employ an indirection through `std::allocator_traits<Alloc>` for creating and destroying objects.

C++20 introduces compile-time allocators support by declaring the corresponding functions **`constexpr`**. However, while such allocators can be used at compile time, the memory allocated by the standard allocator at compile-time must also be released at compile-time, so it is impossible to create regular `std::vector` or `std::string` with content at compile time and later use them at run-time.

C++17 <memory_resource>

`polymorphic_allocator` used by `std::pmr::string` and `std::pmr::vector` employs classes derived from `memory_resource` for dedicated allocation strategies.

- memory resources provided use the decorator pattern to add features
  - `synchronized_pool_resource` - thread safe pool
  - `unsynchronized_pool_resource` - no-sync overhead memory pool
  - `monotonic_buffer_resource` - very fast, releasing everything at destruction
- A global default memory resource relying on `::new` and `::delete` can be obtained
  - `new_delete_resource() noexcept;`
- A memory resource applying the null-object design pattern is also available
  - `null_memory_resource() noexcept;`
  - employ this as a base for a `monotonic_buffer_resource` to limit allocations
  - and for testing out-of-memory situation behavior

Speaker notes

https://en.cppreference.com/w/cpp/memory/synchronized_pool_resource

https://en.cppreference.com/w/cpp/memory/unsynchronized_pool_resource

https://en.cppreference.com/w/cpp/memory/monotonic_buffer_resource

https://en.cppreference.com/w/cpp/memory/null_memory_resource

Polymorphic Memory Resource Example

```
int use_vector_with_pmr_resource(){
  using namespace std::pmr;
  std::array<std::byte,1000> heap{};
  monotonic_buffer_resource memory{
    heap.data()
        ,heap.size()
        ,null_memory_resource()};
  polymorphic_allocator<std::byte> const myalloc { &memory };
  vector<int64_t> v{myalloc};
  try {
    //v.reserve(sizeof(heap)/sizeof(decltype(v)::value_type));
    generate_n(std::back_inserter(v),1000,[i=0]()mutable{return i++;});
  } catch (std::bad_alloc const &) {
    // ignore here intentionally
  }
  return std::accumulate(begin(v),end(v),0);
}
int main() {
  std::cout << use_vector_with_pmr_resource() << '\n';
}
```

https://godbolt.org/z/KMvzThGeh

Speaker notes

https://godbolt.org/z/KMvzThGeh

© 2024 Peter Sommerlad

`std::pmr` or DIY Allocators?

Don’t, unless you can measure the benefit!

- employing dedicated allocators can have huge impact
- passing Allocators as types and eventuelly as objects can clutter the code
- not all details shown here (e.g., copy/move-propagation, scoped_allocator_adapter)
- architecture is important
- small bugs can have huge impact
- consider Allocator support for DIY containers

Speaker notes

I have implemented dedicated allocators in pre-standard C++ in a beneficial way with thread-specific memory pools (no synchronization overhead on allocation and deallocation) where each pool was also managed like a `monotonic_buffer_resource`. But this is something where you really need top be able to measure, because it might be hard to out-do the standard library implementors.

Raw Memory

DIY generic containers can require using raw memory

- arrays of `std::byte` are privileged as raw memory
- raw memory can provide location for objects of other type
  - when contained objects are not `default_constructible` or `default_initializable`
  - when objects are not `movable`.
- reinterpreting (type punning) of (raw) memory as another object is usually not allowed in C++
  - `std::bit_cast` (C++20)
  - `std::construct_at` (C++20) or
  - placement `new` in combination with `std::launder()` are needed

Speaker notes

For backwards compatibility, in addition to `std::byte` also the types `char` and `unsigned char` are privileged to denote raw memory.

https://en.cppreference.com/w/cpp/language/classes#Standard-layout_class

std::bit_cast<TO>(from)

`reinterpret_cast` to access an object's binary representation is often UB
`std::memcpy` of the raw bytes can work

```
#include <bit>
constexpr double f64v = 19880124.0;
constexpr auto u64v = std::bit_cast<std::uint64_t>(f64v);
static_assert( std::bit_cast<double>(u64v) == f64v ); // round-trip

constexpr std::uint64_t u64v2 = 0x3fe9000000000000ull;
constexpr auto f64v2 = std::bit_cast<double>(u64v2);
static_assert( std::bit_cast<std::uint64_t>(f64v2) == u64v2 ); // round-
        trip
```

`bit_cast` works in constexpr context

87

Speaker notes

https://en.cppreference.com/w/cpp/numeric/bit_cast

One allowed reinterpretation is between object pointer types and `std::uintptr_t` by **`reinterpret_cast<>()`**. Don't do it, unless you want to log memory addresses, which can be done anyway without doing so.

© 2024 Peter Sommerlad

Raw (Dynamic) Memory

```
struct demo {
  int i; // might have a hole here
  double d;
};
static_assert(sizeof(demo)> sizeof(int)+sizeof(double),"oops no padding");

// provide storage array
alignas(demo) std::array<std::byte, sizeof(demo)>  buf;

// make_unique allocates aligned correctly
auto buf = std::make_unique<std::byte[]>(sizeof(demo));

// non-initialized (C++20)
auto buf = std::make_unique_for_overwrite<std::byte[]>(sizeof(demo));

// non-initialized (C++20, pod only, otherwise default init)
auto ptr = std::make_unique_for_overwrite<demo>();
```

Initialized object by default

alternatives for providing storage for an object of type demo

Speaker notes

`std::make_unique_for_overwrite` is to allow to allocate a unique_ptr of the right type, but with an default initialized object which might mean uninitialized for types without a constructor. normal `std::make_unique` will value-initialize the object and thus zero objects without a constructor.

© 2024 Peter Sommerlad

Creating an object in raw memory

```
optr = std::construct_at(vptr,ctor-args); // C++20
```

old style placement new

```
optr = new (vptr) T{ctor-args};
```

beware of alignment restrictions of T! 💣

The vptr pointer has “byte” resolution, even if a memory cell is 64 bits, therefore we can address each single byte of the memory, as if the memory cells were of 8 bit size

Speaker notes

Not obtaining the resulting pointer from a placement new, is one of the dark corner cases of the C++ standard. 💣 While the pointer argument to placement-new and its returned value refer to the identical address, only the returned pointer officially refers to the living object, and compilers might make use of that.

https://en.cppreference.com/w/cpp/memory/construct_at

If the result of placement new is not used, one needs to apply `ptr = std::launder(ptr)` to tell the compiler that it must assume that the pointer now refers to a living object. See example at

https://en.cppreference.com/w/cpp/memory/destroy_at

for a use of `std::launder`

© 2024 Peter Sommerlad

Raw with object creation

dynamic memory

```
auto buf{std::make_unique<std::byte[]>(sizeof(demo))};
auto ptr{reinterpret_cast<demo*>(&buf[0])};
ptr = std::construct_at(ptr,42,3.14);
// struct is not destroyed,
// memory released by buf's destructor
```

old style placement new

```
auto buf{std::make_unique<std::byte[]>(sizeof(demo))};
auto ptr{new(buf.get()) demo {42,3.14}};
// struct is not destroyed,
// memory released by buf's destructor
```

stack/static memory

```
alignas(demo)
std::array<std::byte, sizeof(demo)>  buf;
auto ptr{reinterpret_cast<demo*>(&buf[0])};
ptr = std::construct_at(ptr, 42, 3.14);
// struct is not destroyed,
// memory released at end of scope
```

old style placement new

```
alignas(demo)
std::array<std::byte, sizeof(demo)>  buf;
auto ptr{new(buf.data()) demo {42,3.14}};
// struct is not destroyed,
// memory released at end of scope
```


Speaker notes

For types with non-trivial destructors this code is insuficient and will result in UB, because destroying the unique pointer would cause undefined behavior, because the object in the storage is not destroyed before.

© 2024 Peter Sommerlad

Ending the lifetime

```
std::destroy_at(optr); // C++17
```

old style

```
optr->~T(); // explicit destructor call
```

neither release the underlying memory

Speaker notes

https://en.cppreference.com/w/cpp/memory/destroy_at

Raw with object creation and destruction

dynamic memory

```
auto buf{std::make_unique<std::byte[]>(sizeof(demo))};
auto ptr{reinterpret_cast<demo*>(&buf[0])};
ptr = std::construct_at(ptr,42,3.14);
ASSERT_EQUAL(42*3.14 , ptr->i * ptr->d);
std::destroy_at(ptr);
// explicit object destruction
```

old style: explicit destructor call

```
auto buf{std::make_unique<std::byte[]>(sizeof(demo))};
auto ptr{new(buf.get()) demo {42,3.14}};
ASSERT_EQUAL(42*3.14 , ptr->i * ptr->d);
ptr->~demo();
// explicit object destruction
```

stack/static memory

```
alignas(demo)
std::array<std::byte, sizeof(demo)> buf;
auto ptr{reinterpret_cast<demo*>(&buf[0])};
ptr = std::construct_at(ptr,42,3.14);
ASSERT_EQUAL(42*3.14 , ptr->i * ptr->d);
std::destroy_at(ptr);
```

old style: explicit destructor call

```
alignas(demo)
std::array<std::byte, sizeof(demo)> buf;
auto ptr{new(buf.data()) demo {42,3.14}};
ASSERT_EQUAL(42*3.14 , ptr->i * ptr->d);
ptr->~demo();
```


Speaker notes

for your own notes

Correctly Accessing an object in raw memory

`std::launder()` your reinterpret_cast pointers

```
alignas(demo) std::array<std::byte, sizeof(demo)>  buf;
auto ptr = reinterpret_cast<demo*>(&buf[0]);
std::construct_at(ptr,42,3.14); // not using result
auto ptr2 = std::launder(reinterpret_cast<demo*>(&buf[0]));
ASSERT_EQUAL(42*3.14 , ptr2->i * ptr2->d);
```

```
auto buf{std::make_unique<std::byte[]>(sizeof(demo))};
auto ptr{reinterpret_cast<demo*>(&buf[0])};
std::construct_at(ptr,42,3.14);
auto ptr2{reinterpret_cast<demo*>(&buf[0])};
ASSERT_EQUAL(42*3.14 , ptr2->i * ptr2->d);
std::destroy_at(ptr2);
```

—> might need launder(…)

Speaker notes

The need to use std::launder() is contentional in the C++ standardization community. However, my current believe is that you need to launder a pointer received via reinterpret_cast to raw memory. It is not needed, when one uses the pointer returned from placement new or construct_at. Neither is possible in a manager where creation of the object in raw memory is separate from the access to the object. Due to the complex pointer aliasing rules of the C++ abstract machine std::launder is needed to implement access to an object via a pointer that was created by a reinterpret_cast to raw memory. Regardless of the need for launder, the reinterpret_cast of a byte-pointer to an object pointer is only valid, if the corresponding memory is correctly aligned and contains a live object of the correct type.

A wrapper with replacment

```
template<typename T>
struct wrapper{
    //construct in-place
    template<typename...PARAMS>
    explicit wrapper(PARAMS &&...args)
    :value(std::forward<PARAMS>(args)...){}
    T value;
    template<typename...PARAMS>
    void replace(PARAMS &&...args)
    {
      std::destroy_at(&value);
      std::construct_at(&value, std::forward<PARAMS>(args)...);
    }
};
```

https://godbolt.org/z/j5jbWzcnd

Speaker notes

A multi-C++ version supporting implementation is provided in the project SimpleWrapper.

Play with it: https://godbolt.org/z/j5jbWzcnd

Note that this version is more elaborated and gets close to what std::optional is doing.

Where Explicit Lifetime?

DON'T, unless you must and know what you are doing!

mostly for supporting non-regular objects

- generic containers for non-movable objects
  - caution about exception safety!
- object wrappers
  - `std::optional`
- replacing objects without assignment
  - not really, but possible
  - caution about exception safety!

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Exception Safety

Manager Types “should just work”™

Levels of exception safety:

3. no guarantee - UB possible, but fastest
2. basic guarantee - no leaks, invariants preserved
1. strong guarantee - transaction: all or nothing
0. `noexcept(true)`/nothrow guarantee

- generic parameter types can give unpleasant exception properties

Speaker notes

for your own notes

© 2024 Peter Sommerlad

0. `noexcept(true)`

Must be provided for

- destructors (implicit)
- move operations (explicit)
- swap operations (explicit)

other functions with `noexcept(true)` can lead to code generation overhead at call sites, because of `std::terminate()` calls inserted on exceptions

Speaker notes

For inline functions that cannot throw, one might consider `noexcept`, but at the moment, compilers tend to be not optimizing away the code to detect an exception and call `std::terminate()`.

1. strong guarantee

`std::vector::push_back()`

- either successfully appends argument
- or vector stays unchanged in case of an exception
  - caveat: `noexcept(false)` move constructor

what can go wrong?

- allocation fails
- copying/moving new element fails
- copying old elements fails

corollary: never have throwing move operations!

Speaker notes

It is hard to provide the strong guarantee in generic containers. One needs to have a clear understanding and model of which things can go wrong and which not.

For example, after a move operation that throws, you will have no idea if the moved-from object is still in a valid state (=its invariants are preserved).

© 2024 Peter Sommerlad

copy-swap for copy-assignment

manager internal allocation can fail:

```
struct mgr {
//...
mgr& operator=(mgr const &other) & {
  mgr tmp{other}; // copy, might fail, no change
  using std::swap();
  swap(*this,tmp); // is noexcept (hopefully)
}
//...
};
```

copy-swap can achieve strong exception guarantee for copy-assignment

Speaker notes

The copy-swap idiom is only useful for copy-assignment to achieve strong guarantee. It must not be applied to move-assignment, because this can lead to endless recursion, because the default implementation of `std::swap()` will employ move construction and move assignment.

© 2024 Peter Sommerlad

2. basic guarantee

Achieving basic guarantee with 2 managed resources is almost impossible!

tips for achieving basic guarantee:

- manage dynamic memory through `std::unique_ptr`/`std::make_unique`
- manage other resources through a Manager type each (SBRM/RAII)
- never try to manually manage two resources in a single Manager type
- prefer value types over relation types
- stay away from “interesting” invariants, unless managed through RAII
- write lot of tests for generic code with bad behaving types
  - or `static_assert` for well behaved template arguments
- combining well-behaving value types just works

Never define a manager with throwing move!

Speaker notes

Why not always strong guarantee? It might be costly to achieve it. For example, a complex operation with strong guarantee might need to keep twice the memory to store keep the old state around until the new state is fully computed (see copy-swap).

You want to employ efficient move operations to reduce overhead. This works well, when move is noexcept. `std::move_if_noexcept()` will help to select move operation only, when it is noexcept and will otherwise select the copy operation:

```
auto x = std::move_if_noexcept(y);
```

here x obtains the value of y either by copy construction or by move construction, if the latter is defined as noexcept.

Note that `std::move_if_noexcept` is still supporting a throwing move operation if the type is move-only.

I know that managing two resources at onces is hard! I tried for std::experimental::unique_resource and failed first, With help of Eric Niebler, we believe it is now OK, but who knows about all corner cases._

© 2024 Peter Sommerlad

3. no exception safety guarantee

usually not acceptable for library code!

Avoiding tips:

- `static_assert` for required behavior on generic parameters
- Consciously use or implement suitable manager types
  - **NO raw pointers**
- Know your relation types that might dangle
- document, what you cannot (`static_`)`assert`

need for speed must be measurable and justifyable

Speaker notes

In (modern) C++ resource-leaks are not excusable. While C programs can have a hard time with respect to managing resources ownership and cleanup, C++'s deterministic lifetime model, transfer of ownership through move with unique managers, and destructors allow fully resource-safe programming.

Summary

Use `std::vector`
or other suitable standard library containers

- provide move operations only, when they are noexcept

Speaker notes

Please don't show off what you learn this week, Prefer value types and simple solutions over error-prone low-level code.

Exercise 3

exercises/exercise03/

Speaker notes

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

© 2024 Peter Sommerlad

More Templates

Compile-time Code Parametrization

- types (and aliases (C++11))
- functions (and lambdas (C++14))
- variables (C++14)
- concepts (C++20)

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Why constrain templates?

Dedicated Overload/Specialization Selection

- SFINAE
- `if constexpr` - C++17
- `require`, **`concept`** - C++20

Prevent Misuse

- + **`static_assert()`** - C++11

Earlier/Better Compile Error

Speaker notes

While the hope of concepts was to get more terse, user-friendly error messages, this is not necessarily true. However, compilers are attempting to better explain why a template instantiation failed. Nevertheless, this can still lead to a lot of text and it might be hard to spot the actual source of the problem.

© 2024 Peter Sommerlad

SFINAE

Substitution Failure Is Not An Error

- when constexpr if is not enough
- pre-C++20 constraints on functions and templates
  - remove function/select function from overload set
  - select template specialization
- prevents hard errors, when multiple options are possible

Speaker notes

Except for the strange acronym the mechanism can be quite handy to prevent using or even an unsuitable function, when there are alternatives available.

potentially “wrong” syntax must depend on a template parameter (directly or indirectly) and it can occur in one of the following positions:

- template parameter or its default argument
- function type, parameter type or return type
  - here also within expressions in unevaluated contexts (often with `decltype(expr)`)

SFINAE does not apply in function bodies, not even in the body of a lambda expression in an unevaluated context.

This limits its applicability to validity of expressions or the validity of forming a type.

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

C++20 concepts can replace most if not all uses of SFINAE and provide a few more capabilities.

Motivating example

- two overloads of `increment()`
  - non-template with implicit conversion
  - function template with no conversion
- function template is a better match
  - but body requires class type with `.increment()`
  - implicit concept!
- implicit conversion to `unsigned` could work

```
unsigned increment(unsigned i) {
  return i++;
}
template<typename T>
T increment(T value) {
  return value.increment();
}
int main() {
  return increment(42); // error
}
```

https://godbolt.org/z/GEsW9rrEo

Not use template specialisation for functions because overload resolution occurs before than template specialisation

Speaker notes

https://godbolt.org/z/GEsW9rrEo

© 2024 Peter Sommerlad

expression SFINAE example

- two overloads of `increment()`
  - non-template with implicit conversion
  - function template with no conversion
- prevent function template match by using required expression in trailing return type (SFINAE)
- implicit conversion to `unsigned` selected

```
unsigned increment(unsigned i) {
  return i++;
}
template<typename T>
auto increment(T value)
  -> decltype(value.increment()) {
  return value.increment();
}
int main() {
  return increment(42);
}
```

https://godbolt.org/z/K676vPr8n

Speaker notes

The given code is not really a splendid example of SFINAE and just there to give an impression of the mechanics.

For example, using decltype is not feasible, when the function return type is void. It is not scaling well, when multiple dependent expressions exists in the body that need to be constrained. It is not easy to use a compile-time type trait/condition (e.g. `std::is_class_v<T>`)

even if the expression is not providing the actual return type, one can use the trailing-return-decltype trick in pre-C++20 for SFINAE by using the comma operator within the decltype. For example, if the template function increment() should return bool instead, we can use `, false` in the decltype expression:

```
unsigned increment(unsigned i) {
  return i++;
}
template<typename T>
auto increment(T value)
  -> decltype(value.increment(), false) {
  return value.increment();
}
int main() {
  return increment(42);
}
```

https://godbolt.org/z/K676vPr8n

`std::enable_if_t`

`std::is_class_v<T>` checks wether T is a class type.

How to ensure a function template overload is only considered for class types?

- Form a type that is invalid, when it is not a class and use it as
  - template parameter default type
  - function return type
  - function parameter type

`std::enable_if_t<bool , type>`

Speaker notes

std::enable_if_t is defined in terms of std::enable_if. This class template is specialized for true by providing a type alias as a member and empty without such an alias otherwise:

```
template<bool expr, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> {
using type = T;
};

template<bool expr,
         typename T = void>
using enable_if_t = typename enable_if<expr, T>::type;
```

accessing the `::type` member fails, if the compile-time condition provided is false. This forms an invalid type which does not compile, but is SFINAE-friendly.

```
#include<type_traits>
int main() {
    std::enable_if_t<true,int> i;
    //std::enable_if_t<false,int> doesntcompile;
}
```

https://godbolt.org/z/W3xP5xb5W

Applying `std::enable_if_t`

```
template<typename T, typename=SFINAE >
SFINAE increment( SFINAE value) {
 return value.increment();
}
```

Positions:

- template parameter default type
- function return type
- function parameter type

Speaker notes

for your own notes

© 2024 Peter Sommerlad

SFINAE return type

```
template<typename T>
std::enable_if_t<std::is_class_v<T>, T>
increment(T value) {
  return value.increment();
}
```

```
template<typename T>
auto increment(T value) -> std::enable_if_t<std::is_class_v<T>, T>  {
  return value.increment();
}
```

condition `is_class_v<T>` is incomplete concept

Speaker notes

the syntax with trailing return type is necessary, when one needs to check for a valid expression formed from the function arguments.

© 2024 Peter Sommerlad

SFINAE return type `decltype()`

- need trailing return type, when referring parameter

```
template<typename T>
auto increment(T value)
-> decltype(value.increment())  {
  return value.increment();
}
```

- `std::declval<T>()` can form any value in unevaluated context

```
template<typename T>
decltype(std::declval<T>().increment())
increment(T value) {
  return value.increment();
}
```

—> we do not need the actual variable

Speaker notes

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

https://en.cppreference.com/w/cpp/utility/declval

© 2024 Peter Sommerlad

SFINAE on parameter type

often hinders template argument deduction

```
template<typename T>
T increment(std::enable_if_t<std::is_class_v<T>, T> value) {
  return value.increment();
}
```

workaround with extra defaulted parameter:

```
template<typename T>
T increment(T value, std::enable_if_t<std::is_class_v<T>, bool> =false) {
  return value.increment();
}
```


Speaker notes

The first version doesn't compile, because the argument cannot be deduced:

```
../SFINAEwthihEnableIf.cpp:47:3: error: no matching function for call to 'increment'
  increment(c);
  ^~~~~~~~~
../SFINAEwthihEnableIf.cpp:4:10: note: candidate function not viable: no known conversion from 'counter' to 'unsigned int' for 1st argument
unsigned increment(unsigned i) {
         ^

../SFINAEwthihEnableIf.cpp:26:3: note: candidate template ignored: couldn't infer template argument 'T'
T increment(std::enable_if_t<std::is_class_v<T>, T> value) {
  ^
```

Watch out for the space between `bool>` and `=false`. Without it, the code wouldn't compile, because the C++ parser's "max munch" principle, would parse it as `>=` instead of the closing template angle bracket and assignment.

SFINAE on template parameter

this is the only way for class template specializations to use SFINAE

```
template<typename T, typename = std::enable_if_t<std::is_class_v<T>, void>>
T increment(T value) {
  return value.increment();
}
```

- constraining class template constructors
  - template parameter when constructor template
  - constructor parameter based on class template parameter

https://godbolt.org/z/roEKbz5r4

Speaker notes

https://godbolt.org/z/roEKbz5r4

Instead of the default argument with a typename parameter, one can also use the result type of enable_if_t directly (e.g. bool) to specify a non-type-template parameter with a default value.

SFINAE: What for?

- prevent a specific template (partial) specialization
  - better: `static_assert`
- influence overload selection based on type
  - consider `if constexpr` (C++17)
  - tag-dispatch (see `<algorithm>` on `std::iterator_traits`) —> Are now deprecated
  - consider concepts (S++20)
- compute a type trait
  - see detection idiom

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Detection idiom: why

```
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;
  }
};
```

code should be excluded if prefix or suffix are not defined or cannot be output

Speaker notes

The problem to

see also https://en.cppreference.com/w/cpp/experimental/is_detected

Detection idiom - classic trait

```
// detect prefix and suffix static members for output
template<typename U, typename = void>
struct has_prefix : std::false_type {};
template<typename U>
struct has_prefix<U, std::void_t<
    decltype(std::declval<std::ostream&>() << U::prefix)>>
: std::true_type {};
template<typename U, typename = void>
struct has_suffix: std::false_type {};
template<typename U>
struct has_suffix<U, std::void_t<
    decltype(std::declval<std::ostream&>() << U::suffix)>>
: std::true_type {};
}
```

the more-specialized version "wins" if it is well-formed

Speaker notes

For historical reasons C++ type traits have been formed by delegating to `std::true_type` and `std::false_type` that each convert into `bool`

With variable templates that were introduced in C++14, C++17 defined type traits in addition as variable templates, usually by delegating to the `::value` constexpr static data member of `std::bool_constant<bool>`/`std::integral_constant<typename T, T value>`. These types represent individual values.

```
template< class T >
inline constexpr bool is_integral_v = is_integral<T>::value;
```

```
template<typename T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant; // using injected-class-name
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; } // since c++14
};
```

The duality of representing constant values as types, or even types as compile-time values is a key mechanism to meta-programming.

Helper template std::void_t was made available in C++17, but DIY is simple

```
template< class... >
using void_t = void;
```

It is just use to be able to form a (list of) valid type(s), which fails, if one of the arguments is an invalid type.

https://en.cppreference.com/w/cpp/types/void_t

Detection idiom - variable templates

```
template<typename U, typename = void>
constexpr bool has_prefix_v {false}; // default case
template<typename U>
constexpr bool has_prefix_v<U, std::void_t<
    decltype(std::declval<std::ostream&>() << U::prefix)>>
{true};
template<typename U, typename = void>
constexpr bool has_suffix_v {false}; // default case
template<typename U>
constexpr bool has_suffix_v<U, std::void_t<
    decltype(std::declval<std::ostream&>() << U::suffix)>>
{true};
```

variable templates can directly construct the `bool`

Speaker notes

As a nitty detail, the mechanism used here and in the standard library as well is not officially specified in the C++ standard, which was recognized in 2014 but never officially fixed.

see https://en.cppreference.com/w/cpp/language/sfinae

© 2024 Peter Sommerlad

Detection idiom - using variable templates

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


Speaker notes

With variable templates the code is slightly simpler.

Before C++17 one would need 4 overloads and dispatch to the appropriate one by either `true_type` or `false_type`

That is the situation where representing a value as a type is beneficial. However, C++17 `if constexpr` eliminated the need for many simple cases.

© 2024 Peter Sommerlad

Tag dispatch (C++14 or earlier)

```
template <typename U>
class Out{
  static std::ostream& print(std::ostream &out, U const &r, std::true_type, std::true_type)
  {
    return out << U::prefix << r.value << U::suffix;
  }
  static std::ostream& print(std::ostream &out, U const &r, std::true_type, std::false_type)
  {
      return out << U::prefix << r.value;
  }
  static std::ostream& print(std::ostream &out, U const &r, std::false_type, std::true_type)
  {
    return out << r.value << U::suffix;
  }
  static std::ostream& print(std::ostream &out, U const &r, std::false_type, std::false_type)
  {
    return out << r.value;
  }
  friend std::ostream&
  operator<<(std::ostream &out, U const &r) {
    using namespace detail__;
     return print(out, r, has_prefix<U>{}, has_suffix<U>{});
  }
};
```


Speaker notes

When there are more than 2 cases to consider, using individual functions for each **`if constexpr`** will use fewer overloads, there is no need for exponential number of overloads!

© 2024 Peter Sommerlad

Concepts instead of SFINAE

most of what C++20 concepts allow is available in previous versions just with much more and uglier syntax.

- concept is (almost) equivalent to a constexpr bool variable template

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

`requires` *keyword has two uses*

122

Speaker notes

Unfortunately, for historical reasons, the introduction of variable templates and concepts occurred independently and we missed the chance to use bool variable templates as concepts. This lead to the situation that I often define a variable template to define a concept. I might even use a requires expression to define the variable template. I do not know if that is a good or bad practice yet, because concepts themselves can be used as conditions in **`if constexpr`**

requires expression

check if a template parameter supports specific expressions with optionally a constrained result type

```
requires (T t) { { ++t } -> std::same_as<T&>; }
```

is equivalent to

```
std::is_same_v<decltype(++std::declval<T&>()), T&>
```

- produces **`consteval`** `bool` value
  - init of bool variable template
  - definition of **`concept`**
  - condition in **`if constexpr`**

Speaker notes

As we can see, simple requires expressions are a substitute for **`decltype()`** with much simpler syntax.

the parenthesis `()` after the **`requires`** keyword can be used to create “invented” parameters that are only used within the curly braces `{}` to form expressions that must be valid. Each expression is terminated with a semicolon. If the expression must result in a specific type, it can be enclosed in additional braces followed by an arrow `->` specifying a constraint on the result type (similar to the “trailing return type” syntax for lambdas and C++11 **`auto`** return type functions) In addition to expressions one can use **`typename`** to form a type that must be a valid type. Last but not least, one can use `requires` within the body of a requires expression to check for the validity of a concept.

Caution: the body of a lambda expression, even if it is an expression cannot be used to define constraints, only “normal” expressions formed work. This is consistent with the SFINAE restrictions that only work on forming types or syntactical expressions in unevaluated contexts.

requires constraint

concepts can limit templates like SFINAE

```
unsigned increment(unsigned i) {
  return i++;
}
template<typename T>
auto increment(T value)
requires requires (T x) { {x.increment()} -> std::same_as<T>;}
{
  return value.increment();
}
```

`requires requires` *is not a typo!*

https://godbolt.org/z/Kx4nhK3x6

First “requires”: this function has a requirement on its compilation
Second “requires”: specify the requirement

Speaker notes

For the return type constraint, we have to use the binary standard concept `std::same_as`, where the second argument is provided by the expression’s type.

https://godbolt.org/z/Kx4nhK3x6

defining concepts

*take **`requires`** expression and provide a name*

```
template<typename T>
concept incrementable = requires (T x) { {x.increment()} ->
        std::same_as<T>;};
```

can use in requires clause:

```
template<typename T>
auto increment(T value)
requires incrementable<T>
{
  return value.increment();
}
```

```
template<typename T>
requires incrementable<T>
auto increment(T value)
{
  return value.increment();
}
```


Speaker notes

In the requires clause a concept must be provided with its template arguments!

https://godbolt.org/z/oahEn3Paj

multiple concepts/conditions can be combined with the usual logical operators. There is some extra magic about such combined concepts in a requires clause that for me so far didn't differ from my assumptions. It is related with syntactical equivalent of concepts and so-called "subsumption" that identifies needless checks, because on concept is always fulfilled as part of another. This provides a partial order of concepts and allows to select the "most-specific" match to win in overload/specialization selection, when multiple matches occur. If multiple matches with the same best rank occur the ambiguity leads to a compile error. It is also a compile error when multiple matches have incomparable concepts that neither subsume the other. The details are so tricky that I actually cannot remember them from the top of my head or needed them yet in my own code.

see "Partial ordering of constraints" at https://en.cppreference.com/w/cpp/language/constraints

using concepts for shorter syntax

template typename parameter instead of typename

```
template<incrementable T>
auto increment(T value)
{
  return value.increment();
}
```

*function parameter, return type (or variable) with **`auto`***

```
auto increment(incrementable auto value){
  return value.increment();
}
```

concept's template argument is implicit here

Speaker notes

I am not yet a big fan of using the concept name to implicitly define a function template without using the keyword template.

It is much harder to spot the embedded **`auto`**. However, I must confess it is a nice shorthand syntax.

Using `auto` for non-type template parameters is a feature of C++17 that is extended in C++20 by allowing to constrain the type:

```
template<std::integral auto by>
auto increment_by(std::integral auto value)
{
  return value+=by;
}
```

I personally prefer the shorthand syntax only for template parameter definition, because this at least allows to have backward-compatible alternative syntax selected with the preprocessor, or to disable the requires clause of a function template via the preprocessor conditional compilation in pre-C++20 mode.

Concept and backward compatible SFINAE

```
#ifdef __cpp_concepts
template<an_integer TARGET, a_safeint E>
#else
template<typename TARGET, typename E,
        std::enable_if_t<detail_::is_known_integer_v<TARGET>
                     && detail_::is_safeint_v<E>,bool> = false >
#endif
constexpr auto
promote_and_extend_to_unsigned(E val) noexcept
{ ... }
```


Speaker notes

from https://github.com/PeterSommerlad/PSsimplesafeint my safe integer replacement library.

Perfect Match (C++20)

Preventing implicit argument conversion

```
char const &first_char(std::same_as<std::string> auto const &s){
  return s[0]; // no dangling, must have been a std::string argument
}

int main(){
    auto &c = first_char("Hello");// doesn't compile
}
```

https://compiler-explorer.com/z/T186ave95

Speaker notes

Thanks to Jason Turner (@lefticus): https://twitter.com/lefticus/status/1542183060795301888?s=20&t=tggeLV0qNKagwLuYxEDeXA

There are also other ways to prevent implicit conversions, such as deleting the rvalue-reference overload:

```
#include <string>

std::size_t func(std::string const & s)
{
  return s.size();
}
std::size_t func(std::string && s)=delete;

int main()
{
    using namespace std::literals;
  //[[maybe_unused]] auto value = func("Hello World"); // fails to compile
  //[[maybe_unused]] auto value = func("Hello World"s); // fails to compile
  auto const str="Hello World"s;
  [[maybe_unused]] auto value = func(str); // compiles
}
```

However, that would prevent functions that could benefit from move operations.

https://godbolt.org/z/MGxcPc691

Summary: SFINAE and Concepts

Should you strive to constrain your templates?

NO, unless you really need it

- overload selection support is helpful
- disabling a function template overload can help
- guiding template specialization selection can help

Speaker notes

Not fulfilling a required concept can lead to error messages that might be clearer (“concept mismatch/no available overload”) over classical template instantiation errors (“error in …instantiated from … *”) but I haven’t seen a compile error where I personally benefited from concepts.

© 2024 Peter Sommerlad

Exercise 4

exercises/exercise04/

130

Speaker notes

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

© 2024 Peter Sommerlad

C++ Parallelism and Concurrency

- parallelism and concurrency are hard
- even world-class experts get it sometimes wrong
- required synchronizations can make system slower
- problem must suit parallel architecture

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Concurrency Problems

- race conditions
  - check and acting upon result not atomic
- data race: **undefined behavior**
  - concurrent access of shared mutable data
- deadlock: circular blocking waits
- starvation: unfair wake up
- livelock: circular non-blocked waits

Speaker notes

for your own notes

© 2024 Peter Sommerlad

What is a race condition?

- You walk into town and see a nice T-shirt in shop window
- You think about it and decide to buy it
- But you first need to pick up your prescription at the pharmacy, because it closes soon
- When you return to the shop, the T-shirt is no longer on display
- You enter and ask for the T-shirt that was on display
- The shop keeper tells you, that the last one was just sold

Interleaving between the decision based on a condition and the acting up, the condition changed

Speaker notes

Race conditions happen in real life and in computers.

Database systems can attempt to prevent race conditions with “pessimistic locking”, or detect the occurrence of a race condition with “optimistic locking”.

While the pessimistic approach limits concurrency, the later can result in user dissatisfaction through cancelled transactions.

© 2024 Peter Sommerlad

Expensive Synchronisation

- when synchronisation is required it can be **expensive**
- bring all cores to a halt
- stalls cores' pipelines
- synchronize caches
- make writes visible (write through/read through)
- regardless of the mechanism (atomics, mutex)

`volatile` *for sharing data across threads*
DOES NOT WORK

135

Speaker notes

On multi-core systems, mutexes and atomic access can be very expensive compared to regular memory accesses.

© 2024 Peter Sommerlad

Parallelization?

- Suitable Problem?
- Architecture first!
- Minimize need for synchronisation!
- Prove that there are no
  - data races
  - deadlocks
  - other bad behaviour
- Even world-class experts make mistakes!

Speaker notes

To paralellize a problem, one need to map the problem to a suitable architecture first. There exist many architectural/design patterns on how to arrange computation in a parallelizable way. For example: Processing Pipelines/Pipes & Filters, Embarrassingly Parallel, Map-Reduce, …

For real-time applications:

For example, automotive control units use special real-time multi-cores that operate in lock-step to enable bounding instructions and simplify synchronization.

General guidance: separate real-time critical parts from uncritical parts and look out for accidental priority inversion/deadline misses.

© 2024 Peter Sommerlad

C++ standard concurrency

- parallel algorithms
- `async` and futures
- `jthread` (C++20) and `thread`
- `mutex`, `scoped_lock` and `unique_lock`
- `condition_variable`
- `atomic<T>`
- **`thread_local`** storage class specifier
- (C++20 Coroutines (**`co_await`**, **`co_yield`**, **`co_return`**))

Speaker notes

The above lists tries to be sorted by “ease of use”, but is not necessarily that one would use async(), when there is no parallel algorithm.

Unfortunately, the design of higher-level parallelization infrastructure for the C++ standard library was highly contentious and is still in flux. This includes the library support for C++20 Coroutines. Because of the lacking library support and also the still lacking implementation of coroutines in the major compilers, we won’t look at those during this course.

© 2024 Peter Sommerlad

C++ parallel algorithms

- most of `<algorithm>`, new ones in `<numeric>`
- execution policy as additional first parameter (`std::execution::`)
  - `seq`, `par`, `par_unseq`, `unseq` (C++20)
- `par` might start new threads
- `par_unseq`, `unseq` use vectorization
- data elements and operations must be independent for parallelization
- new names in `<numeric>`, e.g.,
  - `accumulate()` becomes `reduce()`
  - `inner_product` -> `transform_reduce`
- best with `vector` and `array` of trivial types

Speaker notes

There are restrictions on data access patterns and what types can be used for parallelization and vectorization.

Not all implementations actually parallelize the algorithms, sequential execution is conforming, even when the execution policy is not `seq`.

© 2024 Peter Sommerlad

parallel algorithm example

```cpp
#include <vector>
#include <iostream>
#include <numeric>
#include <execution> // defines policies
int main()
{
    std::vector<double> xvalues(10007, 2.0), yvalues(10007, 2.0);
    double result = std::transform_reduce(
      std::execution::par,
        xvalues.begin(), xvalues.end(),
        yvalues.begin(), 0.0
    );
    std::cout << result << '\n';
}
```

https://godbolt.org/z/6MMrbr5dW

Speaker notes

https://godbolt.org/z/6MMrbr5dW

Even though execution policy `par` is given, it is not parallelized on major compilers.

```asm
.L7:
        movsd   xmm0, QWORD PTR [rbx+rdx*8]
        mulsd   xmm0, QWORD PTR [rax+rdx*8]
        add     rdx, 1
        addsd   xmm1, xmm0
        cmp     rdx, 10007
        jne     .L7
```

Actually, with Intel Thread-building Block library activated and -O3 code will be parallelized if data is big enough on GCC.

C++ simple `async()`

`std::async()` takes a function object to execute

```
#include <future>
#include <iostream>
#include <cmath>
double do_some_expensive_calculation(double input){
  return std::sqrt(input); // simulate that
}
int main(){
  auto future = std::async(std::launch::async,
                           do_some_expensive_calculation,81.0);
  std::cout << "do some other useful things\n" << std::flush;
  std::cout << "the result is= "<< future.get()<< '\n';
}
```

`std::launch::async` = new thread
`std::launch::deferred` = run on `future.get()`

Speaker notes

for your own notes

C++ `async()` gotchas

- must obtain the future object, even if `void`
  - otherwise, `async()` is synchronous
- starting a new thread is expensive
- pass data by value to the function executed, otherwise dangling or data races can occur
  - future might be returned and scope left
  - `launch::async` starts a new thread
- `future::get()` is one-shot (`std::shared_future`)

Speaker notes

While `async()` was meant to be simple to use, the mixing of deferred execution with concurrent/parallel execution leads to some potentially surprising behavior. For example, the `std::launch::async` policy will start a separate thread to execute the provided function. This causes the returned future object to wait for that thread's termination, either when the value is obtained, or when the future object is destroyed. Not keeping the future returned by async, will cause its destructor to immediately wait for the thread started to terminate. That way it becomes a very expensive sequential call.

Bad `async()` example

```cpp
#include <future> // defines async and future
#include <iostream>
#include <chrono>
unsigned long long fibo_def(unsigned long long n){
  if (n < 1) return 0;
  if (n < 2) return 1;
  auto f1 = std::async( fibo_def,n-1); // obtain future, may be start thread
  auto f2 = std::async( fibo_def,n-2); // obtain future, may be start thread
  return f1.get() + f2.get(); // use future's result
}
int main(int argc, char **argv){
  if (argc < 2 || atoi(argv[1]) <1) {
    std::cout<< "Usage: "<< argv[0] << " number\n";
    return 1;
  }
  unsigned long long n=std::strtoull(argv[1],nullptr,10);
  using namespace std::chrono_literals;
  using Clock=std::chrono::high_resolution_clock;
  Clock::time_point t0 = Clock::now(); // standard millisecond timing
  auto fibn=fibo_def(n);
  Clock::time_point t1 = Clock::now();
  std::cout <<"fibo_def("<<n<<") = " << fibn<<" :" << (t1 - t0)/1ms << "ms\n";
}
```

https://godbolt.org/z/oEKzxWYG5

Speaker notes

https://godbolt.org/z/oEKzxWYG5

© 2024 Peter Sommerlad

Futures and underlying features

- `future<T>` - one-time ticket for a promised result
- `shared_future<T>` - multi-reader ticket
  - requires copyable `T`
  - a future object has `.share()` to create a `shared_future`
- `promise<T>` - shared value holder referred by `future`
- `packaged_task` - abstracts a later invocation, provides a `future`

promise with future can signal state across thread boundaries

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Guaranteeing single execution

multiple threads might try to initialize stuff, that must only be initialized once

- local **`static`** variables are initialized once.
- DIY can use
  - `std::call_once()` - call a function once successfully
  - `std::once_flag` - used to interlock `call_once` calls
  - throwing an exception means unsuccessful

Example cppreference

Speaker notes

for your own notes

© 2024 Peter Sommerlad

C++20 `jthread` vs. `thread`

- both for a thread of execution running the function argument
- `jthread` joins thread in its destructor
- `thread` terminates in its destructor if not joined
- `jthread` provides “stop token” a cooperative interruption mechanism

Speaker notes

for your own notes

© 2024 Peter Sommerlad

simple (and wrong) `thread` example

```cpp
#include <iostream>
#include <thread>
struct UnsynchronizedCounter {
    void increment() & {
        ++value;
    }
    int current() const {
        return value;
    }
private:
    int value{};
};
```

```cpp
int main () {
    UnsynchronizedCounter counter{};
    auto run = [&counter]{ // !!
        for (int i = 0; i < 100'000'000; ++i) {
            counter.increment();
        }
    };
    std::thread t1{run};
    std::thread t2{run};
    t1.join();
    t2.join();
    std::cout << "Counter " << counter.current() << " result\n" ;
}
```

result is (almost always) wrong!

https://godbolt.org/z/K5xdKKd4E

Speaker notes

https://godbolt.org/z/K5xdKKd4E

The shared resource of a counter is passed to the thread function by reference (lambda capture). This almost always indicates a problem, unless the resource is read-only (const/constexpr). Here the mutation by different threads is intentional and causes undefined behavior!

Turning on optimization might flatten the loop and accidentally cause the correct result.

© 2024 Peter Sommerlad

better (not good) `thread` example

```cpp
#include <iostream>
#include <thread>
struct ConcurrentCounter {
  void increment() {
    m.lock();
    ++value;
    m.unlock(); // problematic with exceptions
  }
  int current() const {
    m.lock();
    int const current = value;
    m.unlock(); // problematic with complex flow
    return current;
  }
private:
  mutable std::mutex m{}; // will change in current()
  int value{};
};
```

```cpp
int main () {
  ConcurrentCounter counter{};
  auto run = [&counter]{
    for (int i = 0; i < 100'000'000; ++i) {
      counter.increment();
    }
  };
  std::thread t1{run};
  std::thread t2{run};
  t1.join();
  t2.join();
  std::cout << "Counter "
    << counter.current() << " result\n" ;
}
```

result is correct now, but much slower!

https://godbolt.org/z/h1ez43EPd

Speaker notes

https://godbolt.org/z/h1ez43EPd

Explicitly using the pair of `std::mutex` member functions `lock()` and `unlock()` can break, for example, when an exception is thrown while locked or when the control flow is complex and accidentally a branch returns without unlocking. Therefore, see next slide!

© 2024 Peter Sommerlad

better with `scoped_lock`

```
struct ConcurrentCounter {
    void increment() {
        std::scoped_lock lock{m};
        ++value;
    } // unlocks always
    int current() const {
        std::scoped_lock lock{m};
        return value;
    } // unlocks always
private:
    mutable std::mutex m{};
    int value{};
};
```

```
int main () {
    ConcurrentCounter counter{};
    auto run = [&counter]{
        for (int i = 0; i < 100'000'000; ++i) {
            counter.increment();
        }
    };
    std::thread t1{run};
    std::thread t2{run};
    t1.join();
    t2.join();
    std::cout << "Counter " << counter.current() << " result\n" ;
}
```

result is correct now, and safely locked!

https://godbolt.org/z/zKvn9GqYs

Speaker notes

https://godbolt.org/z/zKvn9GqYs

`std::scoped_lock` always releases the lock in its destructor, regardless if a locking function returns or throws and exception.

In addition `std::scoped_lock` can lock multiple mutexes at once without causing deadlocks, when all such uses use the same set.

© 2024 Peter Sommerlad

better with `atomic<int>`

```cpp
#include <iostream>
#include <thread>
#include <atomic>

struct AtomicCounter {
    void increment() {
        ++value;
    }
    int current() const {
        return value;
    }
private:
    std::atomic<int> value{};
};
```

```cpp
int main () {
    AtomicCounter counter{};
    auto run = [&counter]{
        for (int i = 0; i < 100'000'000; ++i) {
            counter.increment();
        }
    };
    std::thread t1{run};
    std::thread t2{run};
    t1.join();
    t2.join();
    std::cout << "Counter " << counter.current() << " result\n" ;
}
```

For “simple” types and operations use ***`std::atomic<T>`***

https://godbolt.org/z/1Wzsf8qGP

Speaker notes

https://godbolt.org/z/1Wzsf8qGP

For simple values the `std::atomic<T>` wrapper can be used. It is optimized to use the most appropriate mechanism for simple types, such as `int` which also provide “specializations” for operations, such as increment (`operator++`). For user-defined types (which must provide trivially copyable, i.e., not have non-trivial members), the library can chose to use a `std::mutex` to achieve the required atomicity and usually only allows exchanging, reading and writing a value.

The specialized atomics are defined to be compatible/identical to the C11 atomics and thus provide interoperability.

Without further flags, atomic variables provide sequential consistency (`std::memory_order_seq_cst`). However, using atomic variables allows to employ more relaxed memory order flags (relaxed, aquire-release). Use of those flags might allow better performance and more interleaving on some hardware, but is less intuitive and can have surprising results. Correct application in “lock-free” data structures usually calls for a formal proof that all required properties are still correct. Very easy to make things wrong.

“relaxed” atomic for example guarantee data-race-free code (no UB in that respect), but usually do not guarantee visibility of a specific value across threads. incrementing counters can be an example.

Memory-order with `atomic<int>`

```cpp
#include <iostream>
#include <thread>
#include <atomic>

struct AtomicCounter {
    void increment() {
        value.fetch_add(1, relaxed);
    }

    int current() const {
        return value.load(relaxed);
    }
private:
  constexpr static auto
  relaxed{std::memory_order_relaxed};
    std::atomic<int> value{};
};
```

```cpp
int main () {
  AtomicCounter counter{};
  auto run = [&counter]{
    for (int i = 0; i < 100'000'000; ++i) {
      counter.increment();
    }
  };
  std::thread t1{run};
  run(); // at least 100'000'000 seen
  t1.join();
  std::cout << "Counter "
    << counter.current() << " result\n" ;
}
```

Result might be incomplete on some hardware (not Intel)

https://godbolt.org/z/v8osxT33a

Speaker notes

on X86 only sequential consistency is provided by the hardware. PowerPC or ARM for example, have more relaxed atomic operations and thus can deliver faster concurrency at higher risk to get things wrong.

https://godbolt.org/z/v8osxT33a

© 2024 Peter Sommerlad

Higher-level synchronization

Data structures require more than just mutual exclusion

- `condition_variable` - wait on a status protected by a `mutex`
  - use `unique_lock` instead of `scoped_lock`
- `condition_variable_any` - generic CV
- `notify_at_thread_exit` - `notify_all()` CVs

Speaker notes

for your own notes

© 2024 Peter Sommerlad

C++20 Higher-level synchronisation

- `latch` - one-time synchronisation barrier
- `barrier` - cyclic synchronisation barrier
- `binary_semaphore` - semaphore with two states
- `counting_semaphore` - semaphore with non-negative count
- `osyncstream` - wrap `std::ostream` for non-UB, non-intermixed output

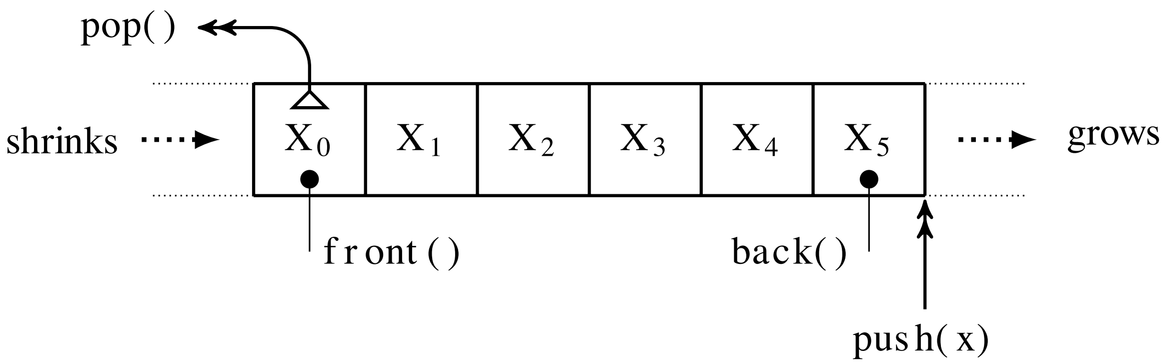
These are still quite low-level building blocks

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Queue



`pop()` needs to wait when `empty()`

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Producer-Consumer Queue

```cpp
#include <condition_variable>
#include <mutex>
#include <queue>
namespace TSQ {
struct ThreadsafeQueue {
    void push(T t) {
        std::scoped_lock const lk { mx };
        q.push(std::move(t));
        notEmpty.notify_one();
    }
    T pop() {
        std::unique_lock lk { mx };
        notEmpty.wait(lk,
          [this] {return !q.empty();});
        T t = std::move(q.front());
        q.pop();
        return t;
    }
```

```cpp
    bool empty() const {
        std::scoped_lock const lk { mx };
        return q.empty();
    }
    void swap(ThreadsafeQueue & other) {
        if (this == &other) {
            return;
        }
        std::scoped_lock const both { mx, other.mx };
        // no need to swap cv or mx
        std::swap(q, other.q);
    }
    friend void swap(ThreadsafeQueue & left,
                     ThreadsafeQueue & other){
        left.swap(other);
    }
private:
    mutable MUTEX mx { };
    std::condition_variable notEmpty { };
    std::queue<T> q { };
};
}
```

154

Speaker notes

Play with it:

https://godbolt.org/z/nWhf487vK

It even supports move-only types:

https://godbolt.org/z/x816qvG1z

© 2024 Peter Sommerlad

Using ThreadSafeQueue

```
int main() {
  using namespace std::this_thread;
  using namespace std::chrono_literals;
  TSQ::ThreadsafeQueue<int> queue { };
  std::thread prod1 { [&] {
    sleep_for(10ms); // demonstration only
    for(int j=0; j < 10; ++j) {
      queue.push(j); yield(); sleep_for(1ms);
    }
  } };
  std::thread prod2 { [&] {
    sleep_for(9ms); // demonstration only
    for(int i=0; i < 10; ++i) {
      queue.push(i*11); yield(); sleep_for(1ms);
    }
  } };
  std::thread cons { [&] {
    sleep_for(15ms); // demonstration only
    do {
      std::osyncstream{std::cout}
        << "consume: "
        << queue.pop() << '\n';
      yield();
    } while (!queue.empty());
  } };
  prod1.join(), prod2.join(), cons.join();
  std::cout << "non-processed elements:\n";
  while (!queue.empty()) {
    std::cout << queue.pop() << '\n';
  }
}
```

Never ***"synchronize"*** *with* `sleep_for()`

https://godbolt.org/z/nWhf487vK

Speaker notes

Namespace `std::this_thread` provides `yield()` and `sleep_for`

Namespace `std::chrono_literals` the `ms` UDL suffix.

`std::osyncstream` wraps any output stream and guarantees 'atomic' output (non interleaving of current output with other threads). For `std::cout` even without `osyncstream` there are no data races, but potential mixed output from different threads. Other output streams (e.g. `ofstream`) that are shared across threads, such wrapping is required

The sleeping is only here to demonstrate variabiltiy in output. Never attempt to use it for solving synchronization issues.

Play with it:

https://godbolt.org/z/nWhf487vK

It even supports move-only types:

https://godbolt.org/z/x816qvG1z

© 2024 Peter Sommerlad

Synchronisation beyond mutex

Mutual exclusion (`std::mutex`) is insufficient

need to wait for other threads' work

condition variables synchronize

- `wait(lock,condition)` - only when mutex held
- internally unlocks mutex
- when notified relocks mutex and checks condition
- other threads need to notify when they fulfil condition
- when fulfilled returns with mutex locked

Speaker notes

Working with `std::condition_variable` requires one to use `std::unique_lock` instead of `std::scoped_lock`, because the condition variable needs a means to unlock and relock the lock.

For locks that do not wrap `std::mutex` one needs to use `std::condition_variable_any`. This might be less performant, because it cannot directly use the corresponding OS synchronisation primitives employed by `std::mutex`.

© 2024 Peter Sommerlad

Single Element Queue

```cpp
struct ThreadsafeExchange {
  void push(T const &t) {
    std::unique_lock lk { mx };
    notFull.wait(lk, [this]() {
      return not q.has_value();
    });
    q.emplace(t);
    notEmpty.notify_one();
  }
  T pop() {
    std::unique_lock lk { mx };
    notEmpty.wait(lk, [this] {
      return q.has_value();
    });
    T t = q.value();
    q.reset();
    notFull.notify_one();
    return t;
  }
```

```cpp
  // don't call when holding a lock!
  bool empty() const {
    std::scoped_lock const lk { mx };
    return not q.has_value();
  }
private:
  mutable MUTEX mx { };
  std::condition_variable notEmpty { };
  std::condition_variable notFull { };
  std::optional<T> q { };
};
```

need 2 condition variables!

we use ***`std::optional`*** *for a bounded queue here*

https://godbolt.org/z/KKaq35e3x

Speaker notes

We now need two condition variables, one marking that there is something to consume in the buffer, the other one to mark there is space in the buffer. With a larger buffer value, this means that both conditions can be met (notFull and notEmpty).

https://godbolt.org/z/KKaq35e3x

trying - really hard

Full synchronisation can deadlock, when opposite partner is gone

```
struct ThreadsafeExchange {
  bool try_push(T const &t) {
    std::scoped_lock const lk { mx };
    if (not q.has_value()){
      q.emplace(t);
      notEmpty.notify_one();
      return true;
    }
    return false;
  }
  template <typename Rep, typename Period>
  bool try_push_for(T const &t, std::chrono::duration<Rep,Period> dur) {
    std::unique_lock lk { mx };
    if (notFull.wait_for(lk, dur, [this]() {
      return not q.has_value();
    })) {
      q.emplace(t);
      notEmpty.notify_one();
      return true;
    } else {
      return false;
    }
  }
```

https://godbolt.org/z/KKaq35e3x

Speaker notes

Be careful to not rely on the empty() member function, because that also acquires the lock and will cause self-deadlock.

https://godbolt.org/z/KKaq35e3x

using a starting latch

prevent threads from premature start

```
std::latch startit{3+1};
// we have 3 threads +main
// start other threads
sleep_for(10ms);
std::cout << "Go go go..."<< std::endl;
startit.arrive_and_wait(); // turn on
```

```
std::jthread prod1 { [&] (std::stop_token stop){
  startit.arrive_and_wait();
//...
std::jthread prod2 { [&] (std::stop_token stop) {
  startit.arrive_and_wait();
//...
std::jthread cons { [&] (std::stop_token stop) {
  startit.arrive_and_wait();
```

https://godbolt.org/z/snYa131ef

multi-use: cyclic `std::barrier`

https://godbolt.org/z/4dzeh64Mj

Speaker notes

https://godbolt.org/z/snYa131ef

https://en.cppreference.com/w/cpp/thread/latch

In addition to the single-pass latch, one can use the cyclic `std::barrier`

https://en.cppreference.com/w/cpp/thread/barrier

This allows multiple synchronisation points and can execute a “completion function” that is called when the barrier opens.

© 2024 Peter Sommerlad

Architecture for Multicore

Introducing parallelism only with clear architecture

- Understand competing goals:
  - latency
  - throughput
  - utilization
- Minimize need for synchronization
- Know usable architectural patterns
  - and choose wisely

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Classic parallelism patterns

- Leader-Followers
- Half-Sync Half-Async
- Pipes and Filters
- Task Farm
- Embarrassing Parallelism

Those are a separate topic

Speaker notes

while I could talk about these things, this would go far beyond the scope of the C++ training and also would only be fruitful after the concrete constraints are much better known.

Common Architectural Features

C++ standard library lacks most

- thread pool(s)
- task (unit of work) abstraction
- task continuations
- synchronized queue(s)
- scheduling mechanism

C++26 may provide those, but still very low-level

162

Speaker notes

In addition to mistakes one can make, choosing the wrong mechanism for the problem at hand, can result in less-than-required performance characteristics.

© 2024 Peter Sommerlad

Threading Summary

- programming multi-threaded code that works correctly is hard
  - even world-class experts make mistakes
- interactive debugging of multi-threaded code often hides synchronization problems
  - core dumps from deadlocks are helpful
- do not attempt multi-threading without clear architecture
  - best to employ architectural patterns or frameworkds
- using parallel algorithms can be helpful
  - unfortunately not all standard libraries actually implement them parallelized
- C++20 coroutines (will) add another dimension to shoot your foot

Speaker notes

© 2024 Peter Sommerlad

Exercise 5

https://github.com/PeterSommerlad/CPPCourseExpert/blo

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Compile-time Computation

`template`

`constexpr` - C++11

`[](){}` - constexpr C++17

`consteval` - C++20

`constinit` - C++20

Speaker notes

While the ability to do compile-time computation in C++98/03 was more-or-less accidental (discovered) through non-type template parameters and static const data members, from C++11 onwards compile-time computability was progressively enhanced over the standard iterations.

© 2024 Peter Sommerlad

Why Comile-time Computation?

- no more `#define MACROS()`
  - plain C++ with real types
- Better run-time performance
  - but compile-times can increase
  - compilers have built-in limits
- limitations lifted over C++ generations
  - “almost everything possible” in C++20

Speaker notes

For the sake of brevity and slide layout I’ll use C-t-C as a short-hand for “compile-time computation” on the following slides.

Evolution C-t-C

C++98/03: class templates with non-type template parameters and `static const` data members, recursion and template specializations for base cases.

C++11: `constexpr` functions (with single `return` statement), variables, literal types, variadic templates

C++14: relaxed `constexpr` function bodies, generic lambdas, `constexpr std::array`, variable templates

C++17: lambdas (implicitly) constexpr, constexpr if, `constexpr` member functions no longer `const`, mutable `array`

C++20: `consteval`, `constinit`, dynamic memory: `vector`, `string`

Speaker notes

With the growing experience in the use of compile-time computation features, design mistakes or deficiencies were made and fixed. For example, first `constexpr` on member functions implied `const`. This was later relaxed to allow computation with literal class types in constexpr functions, but not immediately applied in the standard library, such as std::array. With C++20 further extensions to compile-time computations were allowed, including dynamic memory.

© 2024 Peter Sommerlad

Where is C-t-C guaranteed?

- array bound: `char a[c_t_c]`
- NTTP: `std::array<int,c_t_c>`
- case: `switch(i){ case c_t_c:;}`
- enumerators: `enum{ on=c_t_c}`
- `static_assert(c_t_c,"");` C++11
- `constexpr auto a{c_t_c};`
- constexpr if: `if constexpr(c_t_c)` - C++17
- `noexcept(c_t_c)`, `explicit(c_t_c)` - C++20
- `consteval` functions - C++20
- `constinit` initializer - C++20

Speaker notes

NTTP: non-type template parameter aka value template parameter

Some expressions have always been "core constant expressions", but the contexts were compile-time computation happens in c++ has been extended over the years.

For details see: https://en.cppreference.com/w/cpp/language/constant_expression

unfortunately it is complicated, but that only matters if things don't compile/work as expected.

© 2024 Peter Sommerlad

C++98 C-t-C

value class template parameter with recursion base case through specialization

```
#include <iostream>
template<size_t N>
struct fact{
    static const size_t value=N * fact<N-1>::value;
};
template<>
struct fact<0>{
    static const size_t value=1;
};
char a[fact<5>::value]; // enforce compile-time eval
int main(){
    std::cout << "sizeof a :" << sizeof(a) << '\n';
    std::cout << "factorial(6) :" << fact<6>::value << '\n';
}
```

https://godbolt.org/z/MYnqW4edP

Speaker notes

https://godbolt.org/z/MYnqW4edP

C++11 C-t-C

single-statement (recursive) ***constexpr*** *function*
base case through ***?:***

```
#include <iostream>
constexpr size_t fact(size_t N) {
    return N==0?1:N*fact(N-1);
}
char a[fact(5)]; // enforce compile-time eval
int main(){
    std::cout << "sizeof a :" << sizeof(a) << '\n';
    std::cout << "factorial(6) :" << fact(6) << '\n';
}
```

https://godbolt.org/z/55rG4cb4d

Speaker notes

https://godbolt.org/z/55rG4cb4d

C++14 C-t-C

constexpr *function with loops and local variables*

```
#include <iostream>
constexpr auto fact(size_t N) {
    size_t result{1};
    while(N>0){
        result *= N--;
    }
    return result;
}
char a[fact(5)]; // enforce compile-time eval
int main(){
    std::cout << "sizeof a :" << sizeof(a) << '\n';
    std::cout << "factorial(6) :" << fact(6) << '\n';
}
```

https://godbolt.org/z/GsW7a5bx5

Speaker notes

https://godbolt.org/z/GsW7a5bx5

C++14 variable templates C-t-C

variable template with value template parameter
recursion with base case through specialization

```
#include <iostream>
template<size_t N>
constexpr size_t fact{ N* fact<N-1>};
template<>
constexpr size_t fact<0>{ 1};
char a[fact<5>]; // enforce compile-time eval
int main(){
    std::cout << "sizeof a :" << sizeof(a) << '\n';
    std::cout << "factorial(6) :" << fact<6> << '\n';
}
```

https://godbolt.org/z/9v5EPTczx

Speaker notes

https://godbolt.org/z/9v5EPTczx

C++17 C-t-C

lambdas are literal types and implicit ***`constexpr`***

```
#include <iostream>
constexpr auto fact{ [](size_t N) {
    size_t result{1};
    for(;N>0;--N){
        result *= N;
    }
    return result;
}};
char a[fact(5)]; // enforce compile-time eval
int main(){
    std::cout << "sizeof a :" << sizeof(a) << '\n';
    std::cout << "factorial(6) :" << fact(6) << '\n';
}
```

https://godbolt.org/z/3aPebhjoP

Speaker notes

https://godbolt.org/z/3aPebhjoP

© 2024 Peter Sommerlad

C++20 C-t-C

consteval *enforces compile-time evaluation*
constinit *enforces compile-time initialization*

```
#include <iostream>
consteval auto fact(size_t N) {
    size_t result{1};
    while(N>0){ result *= N--;}
    return result;
}
char a[fact(5)]; // enforce compile-time eval
constinit auto f3{fact(3)}; // non-const!
int main(){
    std::cout << "sizeof a : " << sizeof(a) << '\n';
    std::cout << "factorial(6) : " << fact(6) << '\n';
    std::cout << "++factorial(3) : " << ++f3 << '\n';
    //fact(f3); // doesn't compile
}
```

https://godbolt.org/z/MvnxY7bG1

Speaker notes

with this link you can see the compile error caused by calling a **`consteval`** function with a non-compile-time argument:

https://godbolt.org/z/3oc6YnsYj

Marking a function as **`consteval`** has the benefit over constexpr that this guarantees that the function is evaluated at compile time.

The **`constinit`** specifier is only relevant for variables with static storage duration (=globals) or **`thread_local`** variables that might change during program execution, because otherwise **`constexpr`** would be sufficient, because it implies **`const`** for variables. Because mutable global variables carry the risk of data races, the `constinit` specifier is rarely useful, except for implicitly synchronizing types, such as `std::mutex` or `std::atomic<T>`, when the template parameter type requires non-trivial but **`consteval`** initialization.

C++20 **`consteval`** vs `std::is_constant_evaluated`

- **`consteval`**: function only for C-t-C
- **`constexpr`**: function for C-t-C or run-time
- `std::is_constant_evaluated()`: true if constexpr function used for C-t-C

```
constexpr double power(double b, unsigned x)
{
  if (std::is_constant_evaluated()) {
    double r = 1.0;
    while (x != 0) {
      if (x & 1) r *= b;
      x /= 2;  b *= b;
    }
    return r;
  } else { // run-time lib:
    return std::pow(b, x);
  }
}
```

https://godbolt.org/z/sj9osYGxn

Speaker notes

https://godbolt.org/z/sj9osYGxn

Experiment, what happens if you change the value of the exponent to 1000u in both cases. What are the results?

https://en.cppreference.com/w/cpp/types/is_constant_evaluated

C++23 will introduce a new `if consteval {} else {}` statement that corresonds to `if (std::is_constant_evaluated())`.

https://en.cppreference.com/w/cpp/language/if#Consteval_if

© 2024 Peter Sommerlad

Restrictions C-t-C

some expressions and statements are not allowed at compile-time but they still can occur in constexpr functions

- throwing exceptions
- undefined behavior, e.g. `int` overflow

if called at run-time everything is OK, at compile-time -> compile error

```
constexpr auto fact(long long N) {
    long long result{1}; // signed integer UB on overflow
    while(N>0){ result *= N--;}
    return result;
}
constexpr auto fails{fact(42)};// compile error due to UB
```

https://godbolt.org/z/h1Mar3z5v

Speaker notes

Some of the restrictions can be used to prevent calling a constexpr function at run-time, but in C++20 we can use **`consteval`** for that.

If one is doubting if a piece of code has undefined behaviour, using it in compile-time-evaluation enforces the compiler to create a hard error.

https://godbolt.org/z/h1Mar3z5v

A lot of standard library features became available for compile-time computation over the years, however, often only one release after the corresponding language feature was established. For example, std::array as an aggregate was usable/constructable in constexpr functions quite early on, but could only be manipulated in a constexpr function with C++17, when almost all member functions were made **`constexpr`** (`fill()` and `swap()` became constexpr with C++20).

© 2024 Peter Sommerlad

Types for C-t-C

Literal Type *can be used at compile time*

- built-in (scalar) types, references
- arrays
- class types with at least one constexpr constructor
  - or an aggregate
  - trivial/constexpr destructor
  - and all subobjects of literal type
- User-defined Literals (UDL operators) ease use

Speaker notes

for all the details see:

https://en.cppreference.com/w/cpp/named_req/LiteralType

C++20 relaxed a lot of previous restrictions, especially the allowance of dynamic memory for literal types and thus provides std::vector and std::string to be usable at compile-time. However, values of those types must not live into the run-time.

Before C++20 virtual member functions and destructors couldn't be constexpr.

There are still a few restrictions on what is allowed in constexpr function bodies which seem to become completely lifted in C++23. However, there are still restrictions to what expressions/statements in a constexpr function are allowed to be executed at compile time.

C++20 provides the special function `std::is_constant_evaluated()` that returns true if a constexpr function is actually evaluated at compile time, or false at run time. C++23 will replace that by a special if statement `if consteval {/*compile-time*/} else { /* run-time */}`.

For limits in C++11 and C++14 constexpr functions, please refer to talks by Scott Schurr's excellent talks at CppCon 2015 on constexpr:

C++ constexpr Introduction https://youtu.be/fZjYCQ8dzTc

C++ constexpr Applications https://youtu.be/qO-9yiAOQqc

Example: Literal Type

- constexpr constructor (>=1)
  - or aggregate of literal types
- trivial/constexpr destructor
- constexpr member functions
  - can have non-constexpr
  - only constepxr usable in C-t-C
- can be class template

```
template <typename T>
class Vec3d {
	std::array<T, 3> values{};
public:
	constexpr Vec3d(T x, T y, T z)
		: values{x, y, z}{}
	constexpr T length() const {
		auto sumsq = x() * x() + y() * y() + z() * z();
		return std::sqrt(sumsq);
	}
	constexpr T & x() & {
		return values[0];
	}
	constexpr T const & x() const & {
		return values[0];
	}
	constexpr T & y() & {
		return values[1];
	}
	constexpr T const & y() const & {
		return values[1];
	}
	constexpr T & z() & {
		return values[2];
	}
	constexpr T const & z() const & {
		return values[2];
	}
};
```


Speaker notes

In retrospect it would be beneficial, if like lambdas, any inline function would be considered constexpr, as long as its body allows it. Unfortunately, this is another case where C++ gets its default syntax/default behavior the wrong way around, as for example with **`const`** vs. **`mutable`**

Usage Vec3d

- can be used in constexpr and regular contexts
- non-const member functions can modify the object (C++14)
- constexpr variables are const

```
constexpr bool
doubleEqual(double first,
            double second,
            double epsilon = 0.0001) {
        return std::abs(first - second)
         < epsilon;
}
constexpr Vec3d<double> create() {
        Vec3d<double> v{1.0, 1.0, 1.0};
        v.x() = 2.0;
        return v;
}
constexpr auto v = create();
static_assert(doubleEqual(v.length(),
         2.4495));
int main() {
        //v.x() = 1.0;
        auto v2 = create();
        v2.x() = 2.0;
}
```


Speaker notes

This is a simple example and using std::array is kind of overkill. However, with that one could implement computing the length by employing a standard algorithm:

```
return std::sqrt(std::reduce(begin(values),end(values),T{},[](auto s, auto e){ return s + e*e;}));
```

Constexpr if

compile-time conditional dependent on template parameters

```
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';
  }
}
```

https://godbolt.org/z/4Yf58ndbd

Speaker notes

constexpr if is often the easier solution for simple cases, where older code required to use function template overloads for the special case.

For example like in:

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


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...);
}

int main() {
  int i{42}; double d{1.25}; std::string book{"Lucid C++"};
  printAll(std::cout, i, d, book);
  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');
}
```

play with it:

https://godbolt.org/z/4Yf58ndbd

Lambda Expressions `constexpr`

variable templates can be lambdas or computed by a lambda

- This allows lambdas paramterized by template arguments to the variable template

```
#include<iostream>
template<size_t N>
constexpr auto divides_by =
  [](auto x)->bool{ return 0 == x%N;};
int main(){
    std::cout << std::boolalpha <<
    divides_by<2>(42) << ' ' << divides_by<3>(42) << ' ' <<
    divides_by<4>(42) << ' ' << divides_by<5>(42) << ' ' <<
    divides_by<6>(42) << ' ' << divides_by<7>(42) ;
    return divides_by<7>(42);
}
```

https://godbolt.org/z/zcccsYb71

Speaker notes

https://godbolt.org/z/zcccsYb71

Using a variable template allows to parametrize a named lambda expression

In C++20 lambda expressions can have template parameters, but those can only refer to generic function parameters and provide typenames for referring to the deduced types, e.g., to put constraints on them with requires clauses, or to guarantee a specific type.

Built-in types are potentially evil

some examples of possible undesirable behavior

- `int` - overflow is UB 💣
- `unsigned` - overflow silently wraps
- `x / 0` - is UB 💣
- `unsigned short` promotes to `int`
- `double` implicitly converts to `int`
- `bool` promotes to `int`

none of the built-in types carries semantic meaning

Speaker notes

While using built-in types is very convenient, there is a lot of legacy behavior (from C) that requires utmost programmer attention to not fall into the traps the type system keeps for backward compatibility. Especially problematic are the silent implicit conversions that happen and can have surprising effects, such as loss of precision.

In addition the C++ type system can be employed to actually give semantic meaning to the values we compute with and prevent accidental nonsensical operations to be applied.

Both of these problems can be dealt by creating wrapper types (Whole Type Pattern) that prevent silent conversions and that only provide meaningful. To make such wrapper types as usable as the built-in types, it is important to implement their behavior with **`constexpr`** functions.

`enum` for wrapping integers

signed integer overflow is undefined behavior

integral promotion is a curse

- enum class types
- operator overloading
- concepts

allow to implement wrapping, non-promoting integers:

Simple Safe Integers psssafeint.h

Speaker notes

An example for wrapping integer types to prevent the worst implicit conversions and error prone operations I have provided a corresponding library:

Simple Safe Integers psssafeint.h

using enums as integers

```
// unsigned
enum class ui8: std::uint8_t{};
enum class ui16: std::uint16_t{};
enum class ui32: std::uint32_t{};
enum class ui64: std::uint64_t{};
// signed
enum class si8: std::int8_t{ };
enum class si16: std::int16_t{};
enum class si32: std::int32_t{};
enum class si64: std::int64_t{};
```


Speaker notes

The standard library uses the trick to create an **`enum class`** type without defining an enumeration value for the type `std::byte`.

While all values of the underlying type are valid values for the corresponding enumeration, the enumeration values never implicitly promote or convert.

We don't look at the corresponding operations right now, but how to create corresponding values within the program.

© 2024 Peter Sommerlad

User-defined Literals (UDL)

```
inline namespace literals {
consteval
ui16 operator""_ui16(unsigned long long val) {
    if (val <= std::numeric_limits<std::underlying_type_t<ui16>>::max()) {
        return ui16(val);
    } else {
        throw "integral constant too large"; // trigger compile-time error
    }
}
// etc...
```


Speaker notes

User-defined literal operator overloads are often good candidates for compile-time computation, because they might include range checks of their arguments, which better happen at compile time. employing a non-constexpr allowed operation will trigger a compilation error.

If one has to use pre-C++20 constexpr instead of consteval for such an UDL operator you might get an occasional run-time error instead, but better than silent truncation.

Please note, that we must use round parentheses when constructing the value (`u16(val)`), because the formal parameter of the UDL operator is always an `unsigned long long` and curly braces would prevent narrowing (`u16{val}` doesn't compile.)

© 2024 Peter Sommerlad

Using UDLs

```
using namespace psssint::literals; // required for UDLs

void ui16intExists() {
    using psssint::ui16;
    auto large=0xff00_ui16;
    //0x10000_ui16; // compile error
    //ui16{0xfffff}; // narrowing detection
    ASSERT_EQUAL(ui16{0xff00u},large);
}
```

https://godbolt.org/z/vK3zobM3f

Speaker notes

It is wise to support ADL by defining wrapper types in their dedicated namespace. The a using declaration can import just the type and any functions and operators defined in the associated namespace will be dragged along implicitly.

For UDL operator definitions we have to use a `using namespace` directive to make them usable (or import just the operator with a complicated using declaration: `using psssint::literals::operator""_ui16;`)

https://godbolt.org/z/vK3zobM3f

Simplest Type Wrappers

```
namespace consumption {
struct literGas{
    double value;
};
struct kmDriven {
    double value;
};
struct literPer100km{
    double value;
    friend
    std::ostream &operator<<(std::ostream &out,
        literPer100km x) {
        return out << x.value << " l/100km ";
    }
};
constexpr
literPer100km operator/(literGas l, kmDriven km){
    return {l.value/(km.value/100.0)};
}
```

```
inline namespace literals {
consteval literGas operator"" _l(long double value){
    return literGas{static_cast<double>(value)};
}
consteval literGas operator"" _l(unsigned long long value)
        {
    return literGas{static_cast<double>(value)};
}
consteval kmDriven operator"" _km(long double value){
    return kmDriven{static_cast<double>(value)};
}
consteval kmDriven operator"" _km(unsigned long long
        value){
    return kmDriven{static_cast<double>(value)};
}
}}
int main()
{
    using namespace consumption::literals;
    std::cout << "the car consumes : " << 40_l / 500_km <<
        '\n';
}
```

https://godbolt.org/z/1oedqrq4h

188

Speaker notes

When providing UDL-operators for floating point numbers, we need to create two overloads, one for when the compiler sees a floating point literal (using decimal point or exponent) and one when it is prefixed with just an number sequence, which is parsed as an integer. Otherwise, one has to always use floating point notation.

Note, that it doesn't make sense to define the output operator overload as constexpr, even so it is defined as an inline function, because the iostream library is not (yet?) available in a constexpr form.

https://godbolt.org/z/1oedqrq4h

© 2024 Peter Sommerlad

UDL parameter possibilities

numbers `123_UDL`

- `(unsgined long long)` - integers
- `(long double)` - floating point
- `(char *)` - number without other
- `<char...>()` - raw number, DIY parse

characters `'a'_UDL`

- `(char)` - character (all variations)

strings `"str"_UDL`

- `(char const *, size_t)` - strings (all variations)
- `<S s>()` - C++20 string for `constexpr S(char const *)`

Speaker notes

The raw UDL operators (taking number characters as template arguments) and the C++20 string UDL operators allow to use the UDL argument as template argument, while the other versions make this impossible (you cannot use function parameters as template arguments anywhere)

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

© 2024 Peter Sommerlad

template UDL for ternary numbers

```cpp
#include<iostream>
#include<cstddef>
namespace ternary {
namespace _impl {
consteval unsigned long long
three_to(std::size_t power) {
  auto result{1ULL};
  while(power>0) {
    result*=3;
    --power;
  }
  return result;
}
constexpr bool is_ternary_digit(char c) {
    return c == '0' || c == '1' || c ==
        '2';
}
constexpr unsigned long long
        value_of(char c) {
  return static_cast<unsigned long long>
        (c - '0');
}
```

```cpp
template<std::same_as<char> ...ARGS>
constexpr unsigned long long
value_of(char c, ARGS...digits){
  static_assert(sizeof...(digits));
  return value_of(c)*three_to(sizeof...(digits))
        + value_of(digits...);
}
}
template<char ...Digits>
consteval unsigned long long operator"" _3()
requires (_impl::is_ternary_digit(Digits) && ...)
{
   return _impl::value_of(Digits...);
}
}
int main(int argc, char **argv) {
    using namespace ternary;
    std::cout << "11_ternary is " << 11_3 << '\n';
    std::cout << "02_ternary is " << 02_3 << '\n';
    std::cout << "120_ternary is " << 120_3 << '\n';
//  std::cout << "14_ternary is " << 14_3 << '\n'; //
        compile-error
}
```

https://godbolt.org/z/TsrKzvdsT

Speaker notes

https://godbolt.org/z/TsrKzvdsT

© 2024 Peter Sommerlad

Supporting quantities with different units

A full fledged units library might be too much, but a simple wrapper type too little

```
struct Speed {
  double kph;
};
```

- What if one needs to support m/s or mph?
- one could normalize on one and support conversion factory functions

```
struct Speed {
static Speed from_Kph(double value);
static Speed from_Mph(double value);
static Speed from_mps(double value);
private:
Speed(double);
};
```


Speaker notes

Hard to extend.

One solution: tag-types

```
struct Kph; // just declare
struct Mhp;
struct mps;
template<typename Unit>
struct Speed{
//...
constexpr auto operator<=>(Speed const & other) const =default;
double value;
};
Speed<Kph> local_limit{50};
```

- different Speeed types
- now we can define conversion

Speaker notes

Note, this approach only works well in the small. If you have quantities with many physical units that need to combine in computations, it is best to use/create a full fledged SI units system. For example, Mateusz Pusz' units library https://github.com/mpusz/units

© 2024 Peter Sommerlad

Conversion based on traits

```
template<typename Target, typename Source>
struct ConversionTraits {
constexpr static Speed<Target> convert(Speed<Source> sourceValue) = delete;
}; // default is no conversion possible
template<typename Target, typename Source>
constexpr Speed<Target> speedCast(Speed<Source> const & source) {
  return Speed<Target>{ ConversionTraits<Target, Source>::convert(source)
        };
}
```

- conversion from a speed with a specific trait to another

Speaker notes

as you can see this generic approach doesn't scale well, because we have to define conversions for each combination explicitly.

© 2024 Peter Sommerlad

Conversion Specialization

```
template<typename Same> // partial specialization
struct ConversionTraits<Same, Same> {
constexpr static Speed<Same> convert(Speed<Same> sourceValue) {
  return sourceValue;
} // no conversion needed
};
```

```
static constexpr double mphToKphFactor { 1.609344 };
template<> // full specialization
struct ConversionTraits<tags::Kph, tags::Mph> {
constexpr static Speed<tags::Kph> convert(Speed<tags::Mph> sourceValue) {
  return Speed<tags::Kph>{static_cast<double>(sourceValue) *
         mphToKphFactor};
}
};
```


Speaker notes

Here we can see the limits, for each combination that one wants to support, a corresponding full specialization of the conversion cast needs to be provided.

© 2024 Peter Sommerlad

Using trait-based conversion and comparison

```
template <typename Unit>
bool isFasterThanWalking(Speed<Unit> speed) {
  return velocity::speedCast<Kph>(speed) > Speed<Kph>{5.0};
}
```

- need to implement comparison operations

```
template <typename LeftTag, typename RightTag>
constexpr bool operator==(Speed<LeftTag> const & lhs, Speed<RightTag> const & rhs)
{
  return lhs.value == speedCast<LeftTag>(rhs).value;
}
template <typename LeftTag, typename RightTag>
constexpr auto operator<=>(Speed<LeftTag> const & lhs, Speed<RightTag> const & rhs)
{
  return lhs <=> speedCast<LeftTag>(rhs);
}
```


Speaker notes

C++20 makes implementing comparison operations simpler by only requiring to implement `operator==` and `operator<=>`. With that all other comparison operators are made available.

Summary Compile-time Computation

- `constexpr` most inline functions (makes inline implicit)
  - `is_constant_evaluated`
- `if constexpr` for special cases in function templates instead of overloads
- `consteval` for UDL operators in C++20
- create literal types, esp. for built-in type wrappers
- `constexpr` or `consteval` for constants
- replace older template compile time magic with simpler constexpr/consteval functions
  - limitation that you cannot use function paramter values as template arguments
- `static_assert` - for compile-time unit tests
  - detects possible UB as compile errors!
- implement strong types as literal types

196

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Exercise 6

exercises/exercise06/

Speaker notes

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

© 2024 Peter Sommerlad

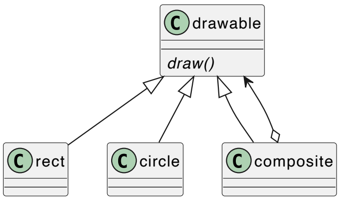
Dynamic Polymorphism without Inheritance

Speaker notes

for your own notes

© 2024 Peter Sommerlad

`virtual` classic




```
struct drawable {
    virtual ~drawable()=default;
    virtual void draw(screen& on)=0;
protected:
    drawable& operator=(drawable&&)=delete;
};
using widget=std::unique_ptr<drawable>;
using widgets=std::vector<widget>;
struct rect : drawable{ ... } ;
struct circle : drawable{ ... };
struct composite : drawable{
  void add(widget w){
    content.push_back(std::move(w));
  }
  void draw(screen &on){
    on << "{ ";
    for(auto &w : content){ w->draw(on); }
    on << " }";
  }
private:
  widgets content{};
};
```

200

Speaker notes

for your own notes

`virtual` usage

```
struct drawable {
    virtual ~drawable()=default;
    virtual void draw(screen& on)=0;
};
using widget=std::unique_ptr<drawable>;
using widgets=std::vector<widget>;
struct rect:drawable{ ... } ;
struct circle:drawable{ ... };
struct composite:drawable{ ... };
```

```
void testComposite(){
    std::ostringstream out{};
    composite c{};
    c.add(std::make_unique<circle>(Radius{42}));
    c.add(std::make_unique<rect>(Width{4},Height{2}));
    c.add(std::make_unique<circle>(Radius{4}));
    c.draw(out);
    ASSERT_EQUAL("{ circle:42rectangle:4,2circle:4 }",
  out.str());
}
```

requires use of base references or (unique) pointers

Speaker notes

If lifetime is guaranteed to be long enough, you could use `std::vector<std::reference_wrapper<draw>>` in the composite implementation.

Type Erasure `std::any`

- `std::function<ret(params)>` can store all function objects that match signature
- `std::any some;` can store any value type
  - muss access type that was last stored
  - might be empty
  - with fixed set: `std::variant`
  - empty state in variant: `std::monostate`

```
void demoAny(){
    std::any some;
    ASSERT(!some.has_value());
    some = 42;
    ASSERT(some.has_value());
    ASSERT_EQUAL(42,std::any_cast<int>(some));
    some = 3.14;
    ASSERT_THROWS(std::any_cast<int>
        (some),std::bad_any_cast);
    some = "anything";
    ASSERT_EQUAL("anything",std::any_cast<char
        const*>(some));
}
```

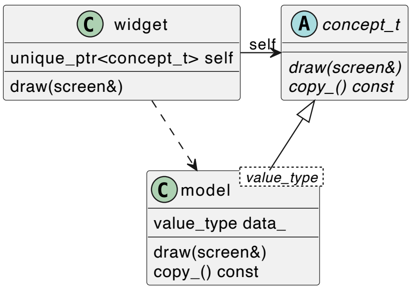

Speaker notes

for your own notes

© 2024 Peter Sommerlad

DIY Type Erasure

- dynamic polymorphism without need for inheritance
- make polymorphic types *regular*
- employs *type erasure*
  - combines inheritance and templates
  - can store arbitrary values, like `std::any`
  - plus a polymorphic interface




```
void testComposite(){
    std::ostringstream out{};
    composite c{};
    c.add(circle(Radius{42}));
    c.add(rect(Width{4},Height{2}));
    c.add(circle(Radius{4}));
    c.add(42);
    c.add("a c string");
    widget w{c};
    draw(w,out);
    ASSERT_EQUAL("... ",out.str());
}
```


Speaker notes

for your own notes

© 2024 Peter Sommerlad

Polymorphic Values with Type Erasure

```
struct widget {
  template<typename T>
  widget(T x) :
      self_(std::make_unique<model<T>>(std::move(x)))
        {
  }
  widget(widget const &x) :
      self_(x.self_->copy_()) {
  }
  widget(widget&&) noexcept = default;
  widget& operator=(widget const &x) & {
    return *this = widget(x);
  }
  widget& operator=(widget&&) & noexcept = default;
  friend void draw(widget const &x, screen &out) {
    x.self_->draw_(out);
  }
  ~widget() = default; // rule of 5, non-virtual
```

```
private:
  struct concept_t { // polymorphic base
    virtual ~concept_t() = default;
    virtual std::unique_ptr<concept_t> copy_() const = 0;
    virtual void draw_(screen&) const = 0;
    concept_t& operator=(concept_t&&) = delete;
  };
  template<typename T>
  struct model: concept_t {
    model(T x) :
        data_(std::move(x)) {
    }
    std::unique_ptr<concept_t> copy_() const {
      return std::make_unique<model>(this->data_); //
        cloning
    }
    void draw_(screen &out) const {
      draw(data_, out); // dispatch to global function
    }

    T data_;
  };
  std::unique_ptr<concept_t> self_;
};
using widgets=std::vector<widget>;
```


Speaker notes

Note that widget is a General Manager class where the defaulted move operations just work, because its sole data member is a unique_ptr.

We get copy-optimization through move for free.

Also recognize that both assigment operators are ensured to only work with lvalues on the left hand side by providing a ref-qualification.

Allowing any Value Types

- provide interface function for existing types
- arbitrary types can be added later that way

```
void draw(std::string s, screen&os){
    os << "string:" << s;
}
void draw(int i, screen &os){
    os << "an_int:" << i;
}
```

- add new types with interface function

```
struct rect{
  rect(Width w, Height h)
  : width{w},height{h}{}
  Width width;
  Height height;
};
void draw(rect const &r, screen& on){
  on << "rectangle:" << r.width
  << "," << r.height;
}
struct circle{
  circle(Radius r) : radius{r}{}
    Radius radius;
};
void draw(circle const &c, screen& on){
    on << "circle:" << c.radius;
}
```


Speaker notes

for your own notes

© 2024 Peter Sommerlad

Composite for type erased widgets

```
struct composite{
    void add(widget w){
        content.emplace_back(std::move(w));
    }
    friend void draw(composite const &c, screen &on){
        on << "{ ";
        for(widget const &drawable:c.content){
            draw(drawable, on); on << ',';
        }
        on << " }";
    }
private:
    widgets content{};
};
```

206

Speaker notes

To prevent superfluous copies, we use `emplace_back(std::move(w))`, but just push_back would also work, because the widget class implements copy operations

© 2024 Peter Sommerlad

`std::variant` fixed set of types

- provide common function interface as before

```
template<typename T>
void draw(T const & o, screen &out){
  out << o;
}
struct composite;
using widget =
std::variant<rect,circle,composite,int,std::string>;
using widgets=std::vector<widget>;
void draw(widget const &w, screen &on);
```

- composite identical as before

- draw variant through `visit()`

```
template<class ... Ts> struct overloaded:
        Ts... {
  using Ts::operator()...;
};
// CTAD to map set of lambdas to object
template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
void draw(widget const &w, screen &on) {
// must provide all overloads for variant
        members
  visit(overloaded {
    [&on](int const &i) { draw(i, on);},
    [&on](std::string const &s) { draw(s,
        on);},
    [&on](rect const &r) { draw(r, on);  },
    [&on](circle const &c) { draw(c, on); },
    [&on](composite const &co) { draw(co,
        on);}
    }, w);
}
```


Speaker notes

`std::variant::visit()` will call the specific lambda based on the type of the stored value. The trick with the overloaded variadic template didn't make it into standardization, but is simple enough to replicate.

It is type safe, if we omit a type we get a compile error. If we do not want to specify a version for each variant, a lambda with an elipses parameter `(...)` can be used as a catch-all option.

Note that through the lambdas, we gain more flexibility for different types, so that a common function is not really needed.

Polymorphism summary

- prefer static polymorphism
- consider relying solely on *regular* value types
  - makes live much simpler
- consciously select dynamic polymorphism mechanism
  - type erasure provides least coupling
  - `virtual` with inheritance strongest coupling
  - `std::variant` often sufficient when alternatives are fixed

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Exercise 7

exercises/exercise07/

Speaker notes

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

© 2024 Peter Sommerlad

Modernizing existing Code

Write Unit Tests (first)!

Don't write C, preprocessor only for `#include`(-guards)

Values over Objects.

Compile-time over Run-time

NO plain arrays or pointers

- Do you have own code to look at?

C++ Introduction C++ Advanced

Speaker notes

Most of the modernization aspects are addressed in the previous courses, but I summarize briefly here.

https://github.com/PeterSommerlad/CPPCourseIntroduction

https://github.com/PeterSommerlad/CPPCourseAdvanced

C++ 20 modules will provide even better modularization. However, the specification has some subtle holes (most of them might be fixed for C++23), and not all major compilers implement modules yet.

No Pointers

- References (as parameter types)
  - for side effects, even on member functions
  - `const&` only for managers and “big” types
- return by value
  - empty `optional<T>` or exceptions to mark errors
- manage memory with `vector`, `string`, containers
  - `unique_ptr` only when other things insufficient

Speaker notes

If you really need optional references, consider using a non-standard optional implementation or `std::optional<std::reference_wrapper<T>>`

© 2024 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>`	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>>`	

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Pointers as Array replacement

<table>
<tr><th></th><th>Old/Unsafe</th><th>Modern/Better</th><th>Alternative</th></tr>
<tr><td>with explicit bound</td>
<td><pre><code>void absarray(int a[], size_t len)
{
    for(size_t i=0; i < len; ++i)
        a[i] = a[i] < 0? -a[i]:a[i];
}</code></pre></td>
<td><pre><code>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];
}</code></pre></td>
<td><pre><code>// 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];
}</code></pre></td></tr>
<tr><td>implicit sentinel nul/nullptr</td>
<td><pre><code>void takecharptr(char *s){
    for (; *s; ++s)
        *s = std::toupper(*s);
}</code></pre></td>
<td><pre><code>void takestring(std::string &s){
  transform(s.begin(),s.end(),
            s.begin(),
      [](char c){
        return std::toupper(c);});
}</code></pre></td>
<td><pre><code>void take(std::string_view s){
    // can not change
}</code></pre></td></tr>
<tr><td>explicit range</td>
<td><pre><code>void absintptrrange(int *b, int *e){
    for (;b != e ; ++b){
        if(*b < 0) *b = - *b;
    }
}</code></pre></td>
<td><pre><code>void absintptrrange(int *b, int *e)
{
  std::transform(b,e,b,
   [](auto i){ return std::abs(i);});
}</code></pre></td>
<td><pre><code>template <typename FWDITER>
void absintarray(FWDITER b, FWDITER e)
{
  std::transform(b,e,b,
    [](auto i){ return std::abs(i);});
}</code></pre></td></tr>
</table>

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Alternatives to plain arrays

```
std::vector<T>
std::array<T,N>
std::string
string_view 💣
span<T> 💣
```


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.

© 2024 Peter Sommerlad

Side-step `virtual`

Unbounded dynamic polymorphism only when needed

- static polymorphism: overloading and templates
- `std::variant`
- type erasure

do not overdo it!

Speaker notes

if existing code uses virtual and inheritance and works well that is OK

© 2024 Peter Sommerlad

Compile-time over run-time

- mark function templates with `constexpr` (or `consteval`)
- templates over class hierarchies
- `static_assert()` over `assert()`
- “test” your code at compile time

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Warnings and Static Analysis

- compile with `-Wall -pedantic -Werror -pedantic-error`
  - `-Wextra` might give too many false positives
  - select further warning candidates
- Run (unit) tests with every build
  - employ `-fsanitize=...` for tests
- look at IDE features for code improvement
  - my past self had many implemented for Cevelop
  - fine tune against false positives
- employ (commercial) static analysis tools
  - look for upcoming new MISRA-C++ guidelines
  - C++ Vulnerabilities

Speaker notes

Don't run static analysis tools as an afterthought. The number of messages can be overwhelming. Each tool might require fine tuning wrt potential false positives. Not every check might be appropriate in your code base, but most violation should have a good reason that goes beyond: "that's just how we implemented it".

https://iso-iec-jtc1-sc22-wg23-cpp.github.io/wg23-tr24772-10-public/

Beware of built-in types

- Use `char` only for text characters
- shorter (unsigned) types promote to `int`
- silent narrowing, widening, sign change happens

Speaker notes

for your own notes

© 2024 Peter Sommerlad

`enum` for wrapping integers

signed integer overflow is undefined behavior

integral promotion is a curse

- enum class types
- operator overloading
- concepts

allow to implement wrapping, non-promoting integers:

Simple Safe Integers psssafeint.h

using enums as integers

```
// unsigned
enum class ui8: std::uint8_t{ tag_to_prevent_mixing_other_enums };
enum class ui16: std::uint16_t{ tag_to_prevent_mixing_other_enums };
enum class ui32: std::uint32_t{ tag_to_prevent_mixing_other_enums };
enum class ui64: std::uint64_t{ tag_to_prevent_mixing_other_enums };
// signed
enum class si8: std::int8_t{ tag_to_prevent_mixing_other_enums };
enum class si16: std::int16_t{tag_to_prevent_mixing_other_enums};
enum class si32: std::int32_t{tag_to_prevent_mixing_other_enums};
enum class si64: std::int64_t{tag_to_prevent_mixing_other_enums};
```

User-defined Literals (UDL)

```
inline namespace literals {
consteval
ui16 operator""_ui16(unsigned long long val) {
    if (val <= std::numeric_limits<std::underlying_type_t<ui16>>::max()) {
        return ui16(val);
    } else {
        throw "integral constant too large"; // trigger compile-time error
    }
}
// etc...
```


Using UDLs

```
using namespace psssint::literals; // required for UDLs

void ui16intExists() {
    using psssint::ui16;
    auto large=0xff00_ui16;
    //0x10000_ui16; // compile error
    //ui16{0xfffff}; // narrowing detection
    ASSERT_EQUAL(ui16{0xff00u},large);
}
```

Traits and Concepts

```
template<typename T>
using plain = std::remove_cvref_t<T>;
template<typename T>
concept an_enum = std::is_enum_v<plain<T>>;
// from C++23
template<an_enum T>
constexpr bool
is_scoped_enum_v = !std::is_convertible_v<T, std::underlying_type_t<T>>;
template<typename T>
concept a_scoped_enum = is_scoped_enum_v<T>;
```


Detection Idiom with Concept

```
template<typename T>
constexpr bool
is_safeint_v = false;
template<a_scoped_enum E>
constexpr bool
is_safeint_v<E> = requires {
  E{} == E::tag_to_prevent_mixing_other_enums;
} ;
template<typename E>
concept a_safeint = is_safeint_v<E>;
```

Testing Detection Idiom

```
namespace _testing {
using namespace psssint;
static_assert(is_safeint_v<ui8>);
static_assert(is_safeint_v<ui16>);
static_assert(is_safeint_v<ui32>);
static_assert(is_safeint_v<ui64>);
static_assert(is_safeint_v<si8>);
static_assert(is_safeint_v<si16>);
static_assert(is_safeint_v<si32>);
static_assert(is_safeint_v<si64>);
enum class enum4test{};
static_assert(!is_safeint_v<enum4test>);
static_assert(!is_safeint_v<std::byte>);
```


Meta Programming for Promotion

```
template<typename E>
using ULT=std::conditional_t<std::is_enum_v<plain<E>>,
             std::underlying_type_t<plain<E>>, plain<E>>;
template<typename E>
using promoted_t = // will promote keeping signedness
    std::conditional_t<
    (std::is_unsigned_v<ULT<E>> && (sizeof(ULT<E>) < sizeof(unsigned))),
    unsigned, ULT<E>>;
template<a_safeint E>
constexpr auto
to_int(E val) noexcept { // promote keeping signedness
    return static_cast<promoted_t<E>>(val);
}
```

Testing Same-sign Promotion

```
static_assert(std::is_same_v<unsigned,decltype(to_int(1_ui8)+1)>);
static_assert(std::is_same_v<unsigned,decltype(to_int(2_ui16)+1)>);
static_assert(std::is_same_v<int8_t,decltype(to_int(1_si8))>);
static_assert(std::is_same_v<int16_t,decltype(to_int(2_si16))>);
```


Concept for limiting integral

```
template<std::integral T>
constexpr bool
is_integer_v =  std::is_same_v<uint8_t, T> ||
                std::is_same_v<uint16_t, T> ||
                std::is_same_v<uint32_t, T> ||
                std::is_same_v<uint64_t, T> ||
                std::is_same_v<int8_t, T> ||
                std::is_same_v<int16_t, T> ||
                std::is_same_v<int32_t, T> ||
                std::is_same_v<int64_t, T>;

template<typename T>
concept an_integer = is_integer_v<T>;
```

Meta Programming for Conversion

```
template<an_integer T>
constexpr auto
from_int(T val) {
    using std::is_same_v;
    using std::conditional_t;
    struct cannot_convert_integer{};
    using result_t =
        conditional_t<is_same_v<uint8_t,T>, ui8,
         conditional_t<is_same_v<uint16_t,T>, ui16,
          conditional_t<is_same_v<uint32_t,T>, ui32,
           conditional_t<is_same_v<uint64_t,T>, ui64,
            conditional_t<is_same_v<int8_t,T>, si8,
             conditional_t<is_same_v<int16_t,T>, si16,
              conditional_t<is_same_v<int32_t,T>, si32,
               conditional_t<is_same_v<int64_t,T>, si64, cannot_convert_integer>>>>>>>>;
    return static_cast<result_t>(val);
}
```


Directed Conversion

```
template<a_safeint TO, an_integer FROM>
constexpr TO
from_int_to(FROM val) {
    using std::is_same_v;
    using std::conditional_t;
    using result_t = TO;
    if constexpr(std::is_unsigned_v<std::underlying_type_t<result_t>>){
        if (val <= std::numeric_limits<std::underlying_type_t<result_t>>::max()) {
            return static_cast<result_t>(val);
        } else {
            throw "integral constant too large";
        }
    } else {
        if (val <= std::numeric_limits<std::underlying_type_t<result_t>>::max() &&
            val >= std::numeric_limits<std::underlying_type_t<result_t>>::min()) {
            return static_cast<result_t>(val);
        } else {
            throw "integral constant out of range";
        }
    }
}
```

Testing `from_int` Conversion

```
static_assert(1_ui8 == from_int(uint8_t(1)));
static_assert(42_si8 == from_int_to<si8>(42));
//static_assert(32_ui8 == from_int(' ')); // does not compile
//static_assert(1_ui8 == from_int_to<ui8>(true)); // does not compile

void checkedFromInt(){
    using namespace psssint;
    ASSERT_THROWS(from_int_to<ui8>(2400u), char const *);
}
```


Output Operator

```
template<a_safeint E>
std::ostream& operator<<(std::ostream &out, E value){
    out << +to_int(value); // + triggers promotion and prevents char
    return out;
}
```

concept `a_safeint` prevents using it for other types

```
std::ostream& operator<<(std::ostream &out, a_safeint auto value){
    out << +to_int(value); // + triggers promotion and prevents char
    return out;
}
```

I prefer `template<a_concept T>` over `(a_concept auto`

Arithmetic Operators

```
template<a_safeint E, a_safeint F>
constexpr auto
operator+(E l, F r) noexcept
requires same_sign<E,F> {
  using result_t=std::conditional_t<sizeof(E)>=sizeof(F),E,F>;
  return static_cast<result_t>(
           static_cast<ULT<result_t>>(
             to_uint(l)
             + // use unsigned op to prevent signed overflow, but wrap.
             to_uint(r)
           ) );
}
```


Wrapping Simple Safe Integers

- Free to use and download
- requires C++20
  - a C++17 backport is available
- Target audience:
  - embedded developers
  - safety critical systems developers
- best if regulare integer types are prevented by static analysis

https://github.com/PeterSommerlad/PSsimplesafeint

Strong Types over built-ins

- Beware of implicit conversions and integral promotion
- Model your system with strong types

C++ Advanced Strong Types

- Consider a units library for physical dimensions

Speaker notes

for your own notes

© 2024 Peter Sommerlad

Exercise 8

- exercises/exercise08/

Speaker notes

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

© 2024 Peter Sommerlad

Done...

Feel free to contact me @PeterSommerlad@mastodon.social (🦣) or peter.cpp@sommerlad.ch in case of further questions

Speaker notes

for your own notes

© 2024 Peter Sommerlad