What Classes We Design and How

What Classes We Design and How

CPPP 2021-12-01

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad

Slides:

https://github.com/PeterSommerlad/talks_public/tree/master/CPPParis/2021

My philosophy

Less Code

=

More Software

What C++-objects model?

Roles:

C++ specific:

What C++-objects model?

Roles:

- **Value** - what

C++ specific:

What C++-objects model?

Roles:

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

C++ specific:

What C++-objects model?

Roles:

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

C++ specific:

What C++-objects model?

Roles:

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

C++ specific:

- **Manager** - clean up

Speaker notes

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

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

Roles of C++ objects (V)

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

C++ specific:

- Manager - clean up

What is a Value 💰?

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

What is a Value 💰?

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

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

Speaker notes

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

outside time  and space 

a value can have different representations

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

behavior is independent of representation and location

a value object is valid independent of other entities

Speaker notes

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

Value Semantics in C++

property of a type

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

Speaker notes

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

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

Values and `const` in C++

immutability is a means to enforce value semantics

- `shared_ptr<T const>` has value semantics

But, do not add `const` to member variables needlessly getting it right and efficient is tricky

See Juanpe Bolívar's CPPCon 2017 talk & immer lib

Speaker notes

obtaining value semantics through immutable types is possible, but often requires sophisticated implementation techniques to keep it efficient. See for example https://sinusoid.es/immer/ (or the CPPCon 2017 talk)

Roles of C++ objects (S)

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

C++ specific:

- Manager - clean up

What is a Subject📌?

*I choose **subject** over object*

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

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

Speaker notes

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

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

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

Polymorphic subject types

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

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

other means for dynamic polymorphism not shown today

Speaker notes

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

Roles of C++ objects (R)

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

C++ specific:

- Manager - clean up

What is a Relation 🔗👉?

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

Speaker notes

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

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

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

“Fernwirkung” 🕹️

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

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

Technicalities of Relation Types 🔗

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

Speaker notes

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

Roles of C++ objects (M)

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

C++ specific:

- **Manager** - clean up

Speaker notes

actually a Manager object is one that is behaving not like a typical business manager, but more like a janitor or well-behaved dog owner: it cleans up the mess, when everything is done.

What is a Manager 👔?

*Manage a **single** resource*

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

Technicalities of Managers 👔

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

Technicalities of Managers 

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

never define a destructor with an empty body

Technicalities of Managers 

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

never define a destructor with an empty body

exceptional cases might manage an invariant and not a resource

Speaker notes

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

Kinds of Manager Types

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

Why?

think about object roles and class kinds

- roles of sub-objects (bases/members) influence
- defining class types correctly can be overwhelming

Speaker notes

unfortunately, roles can overlap for instances of a class

Defining a class: too many options?

- `~T()`
- `T(T const &)`
- `T& operator=(T const&) &`
- `T(T&&) noexcept`
- `T& operator=(T&&) & noexcept`

- public
- protected
- private

- not declared
- `=default;`
- `=delete;`
- with non-empty body

plus all the different spelling options for copy/move

What special member functions we get

What you get

What you write	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted (!)	defaulted (!)	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted (!)	not declared	not declared
copy assignment	defaulted	defaulted	defaulted (!)	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Howard Hinnant's Table: https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf
Note: Getting the defaulted special members denoted with a (!) is an unfixable bug in the standard.

Sub-object influence

- Value: fine ❤️
- Relation:
  - contagious 🦠, or
  - Manager
- Polymorphic (base)
  - fine, when base well defined 💕
- Scoped Manager: contagious 🦠
- Unique Manager: contagious 🦠
- General Manager = Value: fine ❤️

How to define a class!

Don't declare any of the special members

This is called the Rule of Zero (RoZ)

… unless you must

RoZ: What the Core Guidelines say

C.20

If you can avoid defining default operations, do.

Reason

It's the simplest and gives the cleanest semantics.

Simplicity rules!
but rationale sounds a bit weak

Rule of Zero (RoZ)

Implement your classes in a way, that compiler-provided default implementations just work

Even defining special member functions with `=default;` or `=delete;` can change overload resolution, being an aggregate or trivial, and thus behavior or compilability.

exception for `=default;`

you should resurrect a default constructor or define a virtual destructor as `=default;`

When RoZ ill-suited ?

A class that needs to define a destructor

- This was the cause for Scott Meyer's Rule of Three
- And in most cases still is for non-RoZ
  - An odd manager for an invariant (where the default destructor is fine)
    - non-local invariant, or
    - with internal references (= local invariant)

Speaker notes

I still haven't found such strange behaving Manager types in real world code, where it actually had to exist like that.

Polymorphic Base Classes

Deleting via pointer to base of a derived object is ***undefined behavior*** *unless base has* `virtual` *destructor*

C.35

A base class destructor should be `public` and `virtual`, or `protected` and non-`virtual`

Copy will slice objects via base class references

C.67

A polymorphic class should suppress copy/move

Speaker notes

Slicing is just one aspect, why copying should be prevented. The use of relationship objects to access and object should not degenerate to copying, because with the copy, one loses the identity of the original and in the case of slicing even its dynamic type.

How to prevent copying?

Making a class T non-copyable:

```
T& operator=(T&&)=delete;
```

What you get

What you write	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted (!)	defaulted (!)	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted (!)	not declared	not declared
copy assignment	defaulted	defaulted	defaulted (!)	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Howard Hinnant's Table: https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf
Note: Getting the defaulted special members denoted with a (!) is a bug in the standard.

Rule of DesDeMovA 🙋

Prevent a class copy and move with less code

DesDeMovA

Rule of if
Destructor defined
Deleted
Move **A**ssigment

No need to resurrect default constructor

For polymorphic base classes and scoped managers

Polymorphic Base Class

public virtual destructor **`=default;`**
and move assignment **`=delete;`**

```
struct OOBase {
  virtual void somevirtualmember();
  virtual ~OOBase() = default;
  OOBase& operator=(OOBase &&other) = delete;
};
```

least amount of code and distinct to prevent confusion

DesDeMovA

Rule of if
Destructor defined
Deleted
Move **A**ssigment

Managers 

C++ deterministic object lifetime and destructors

Destructor for RAII

Resource-acquisition-is-initialization
Scope-based Resource Management (SBRM)

Destructor with body defined

- Constructor acquires resource
- Destructor releases resource
- need to care about copying/moving (next slide)
- **Do not manage multiple resources at once**
  - I tried with `unique_resource` p0052 and it is too error prone

Speaker notes

*__NEVER__ define a destructor with an empty body. This is not only superfluous code, but also suppresses the compiler-provided move operations, so depending on your class' sub-objects actually a pessimization.

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 } 

```
struct Scoped {
  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>`.

Unique Manager 🏝️

```
class Unique {
    std::optional<Resource> resource;
    void release() noexcept;
public:
    Unique() = default;
    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

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

```
Unique::Unique(Unique &&other) noexcept
:resource{std::move(other.resource)}{
    other.resource.reset();
}
Unique&
Unique::operator=(Unique &&other) & noexcept {
    if (this != &other) {
        this->release();
        std::swap(this->resource, other.resource);
    }
    return *this;
}
```

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

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.

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

General Manager 💰

```
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

General Manager Discussion 💰

*Provide **value semantics** for a resource without*

Expert-level coding, explicit clean-up

Resource must be copyable/replicable

It might be simpler to reuse existing GM types

Take aways 🍔🍟: Roles

create non-value types only with consideration

- deviate from value semantics only when needed
  - polymorphic bases
  - scoped and unique managers
- handle relation objects with care
  - especially when disguised as class types
  - as function parameter types fine

Speaker notes

wrt relation object as parameters: don't make them parameter types for coroutines or thread lambdas. Those will either lead to data races or potential dangling.

Returning a relation object is necessary and possible, but requires close scrutiny to not unnecessarily keep hold of it across statements that invalidate it, e.g., by destroying its target subject.

Take aways 🍔🍟: Special Members

Rule of Zero Rulez

Take aways 🍔🍟: Special Members

Rule of Zero Rulez

Never define a destructor with an empty body

Take aways 🍔🍟: Special Members

Rule of Zero Rulez

Never define a destructor with an empty body

`=default` virtual destructor

Take aways 🍔🍟: Special Members

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.

What was missing?

How to cope with Relation objects? 🚒🧯 🦸‍♀️

- DANGs = potentially dangling types 🧨🔥🎇

What was missing?

How to cope with Relation objects? 🚒🧯 🦸‍♀️

- DANGs = potentially dangling types 🧨🔥🎇

How to encapsulate virtual?
Alternatives for run-time polymorphism?

- `variant<A,B,C>`, Envelope-Letter pattern

Questions & Contact

Peter Sommerlad
peter.cpp@sommerlad.ch
@PeterSommerlad
https://sommerlad.ch

Slides:

https://github.com/PeterSommerlad/talks_public/tree/master/CPPParis/2021