C++ Class Natures in their Canonical Form and how to find them

CPPCon 2019
Prof. Peter Sommerlad
@PeterSommerlad
peter.cpp@sommerlad.ch

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Canonical Form

=

standard way of representation

```
class Example {
public:
  Example();
  virtual ~Example();
  Example(const Example &other);
  Example(Example &&other);
  Example& operator=(const Example &other);
  Example& operator=(Example &&other);
};
```

Really canonical?

```
class Naughty {
public:
  Naughty();
  ~Naughty() noexcept(false);
  Naughty(Naughty &other);
  Naughty(Naughty const &&other) noexcept(false);
  void operator=(Naughty &other) const;
  Naughty& operator=(Naughty const &&other) const noexcept(false);
};
```

What is wrong here?

```
Example::Example(const Example &other)
:Base{other}
,member{other.member} {
// empty
}
```

Easy to forget

should we DIY that

```
void Naughty::operator =(Naughty &other) const {
  other.val = 4;
}
```

DO NOT DO THAT

```
class Good {
public:
  Good() = default;
  ~Good() = default;
  Good(const Good &other) = default;
  Good(Good &&other) noexcept = default;
  Good& operator=(const Good &other) = default;
  Good& operator=(Good &&other) noexcept = default;
};
```

all that spelling...

```
struct Better {
};
```

Less Code

=

More Software

Remember: What Special Member Functions Do You 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.

Class Natures

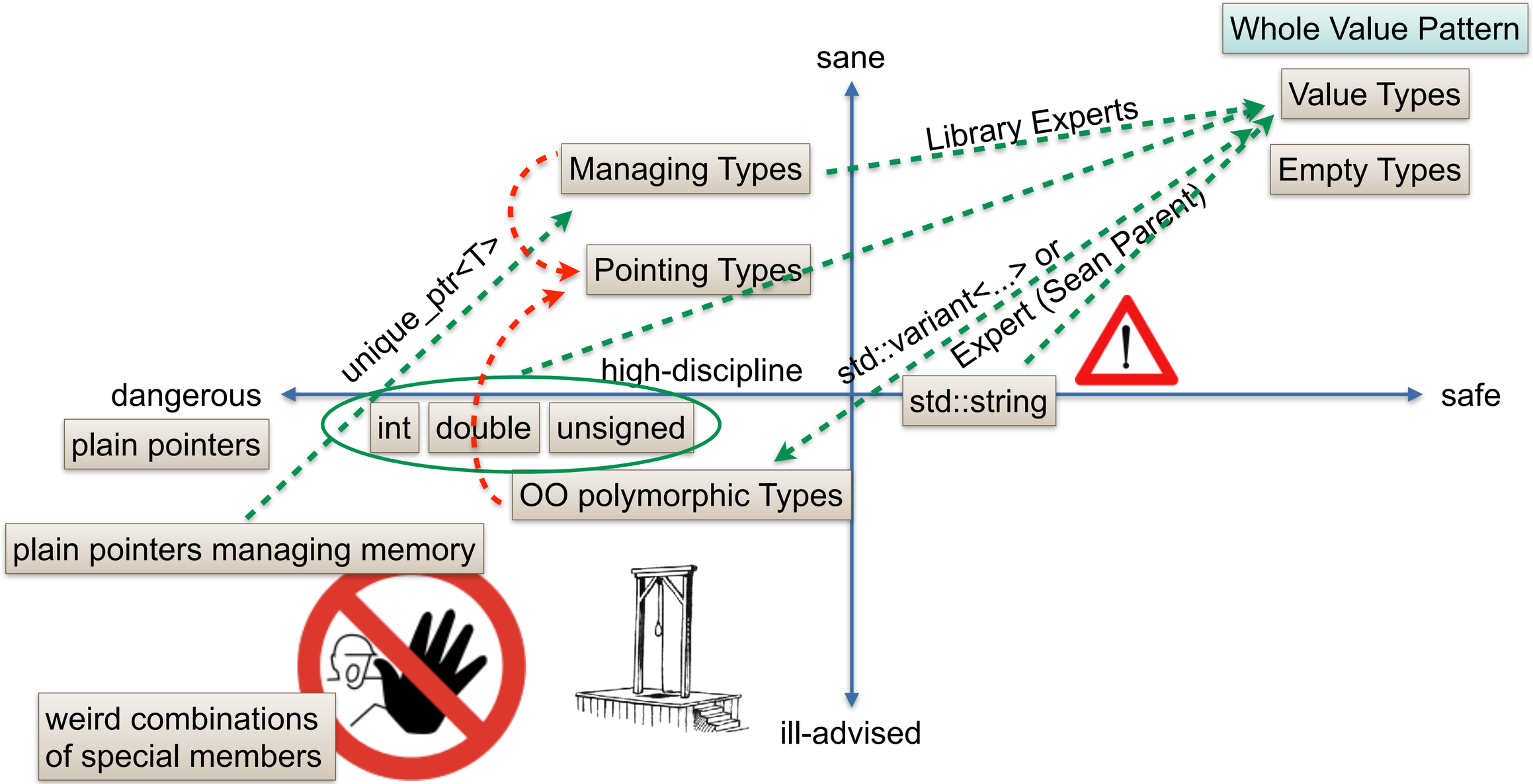
nature: 🏞️🌅🏜️⛰️🏝️🌈🌳

kind 🧬

character 🎎

essence 🔮

Last year...



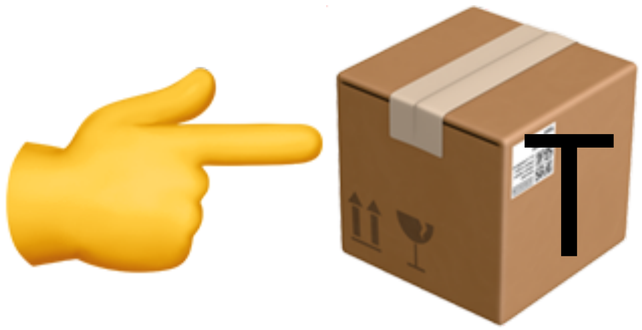
Can we make that better to see all Class Natures?

value

polymorphic

designating

managing



T*

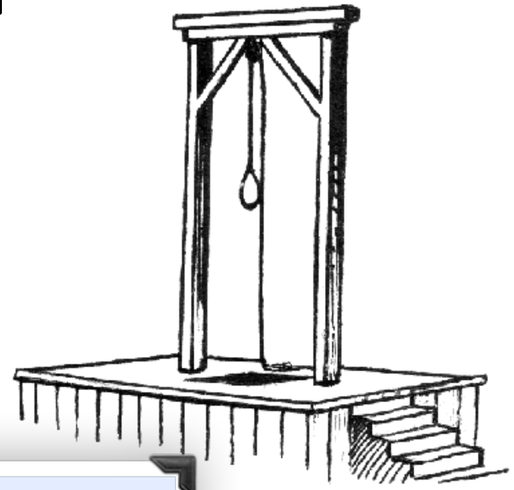
T&

iterators

string_view

views

span<T>



Validity depends on other object not managed by it

value
polymorphic
designating
managing

as:
member
base
parameter
return

value

polymorphic

designating

managing

as:

member ✅

base ❓🔌

parameter ✅

return ✅

value

```
template<typename T>
struct Stack: private std::vector<T>{
  using base=std::vector<T>;
  using std::vector<T>::vector;
  using base::empty;
  void push(T const &i) {
    base::push_back(i);
  }
  T const & top() const {
    return base::back();
  }
  void pop() {
    base::pop_back();
  }
};
```

as:

member ✅

base ❓🔌

parameter ✅

return ✅

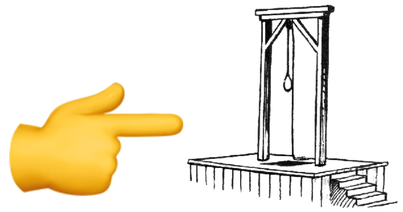
value

polymorphic

designating

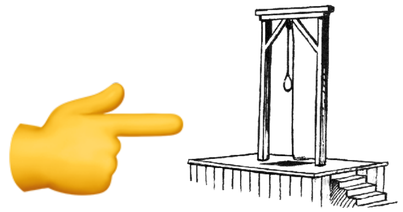
managing

as:

member 

base (OO) ✅

parameter 👉

return 

value

polymorphic

designating

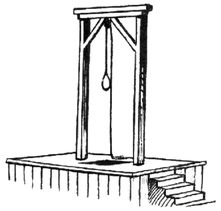
managing

as:

member 

base 

parameter 

return 

it depends!

value

polymorphic

designating

managing

as:

member ?

base ❌

parameter 👉

return ?

value 💰

polymorphic

designating

managing

Canonical Forms?

value 💰

polymorphic

designating

managing

(semi)regular

copyable

~~unique~~

proper

~~shared~~

All defaults just work


```
struct SimplestValue{
  int val;
};

struct Value {
  Value() = default;
private:
  SimplestValue member{};
};
```

Often overused

polymorphic

designating

managing

~~regular~~

non-copyable

unique (_ptr)

improper

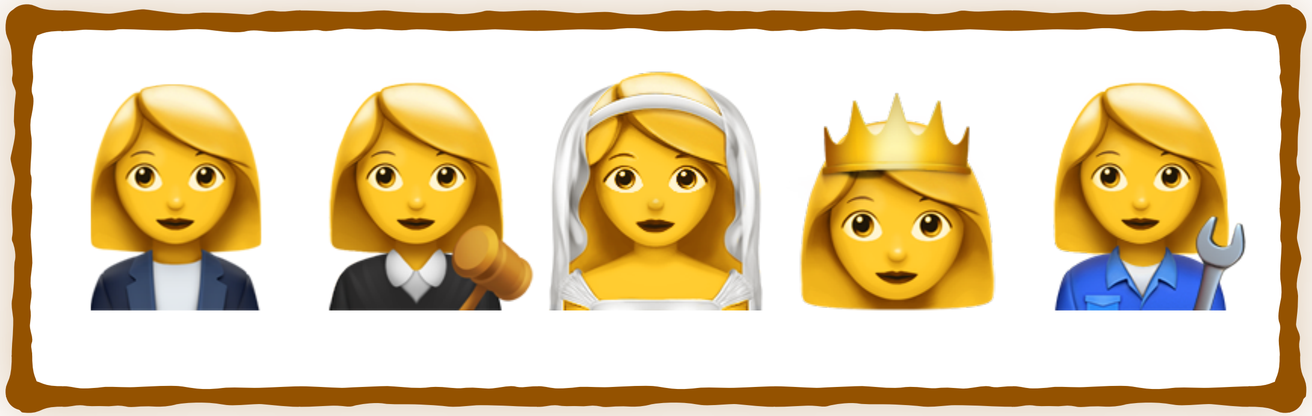
shared (_ptr)

```
class OOBase {
public:
  virtual ~OOBase() = default;
  OOBase& operator=(OOBase &&other) = delete;
};
```

DesDeMovA

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

value



polymorphic

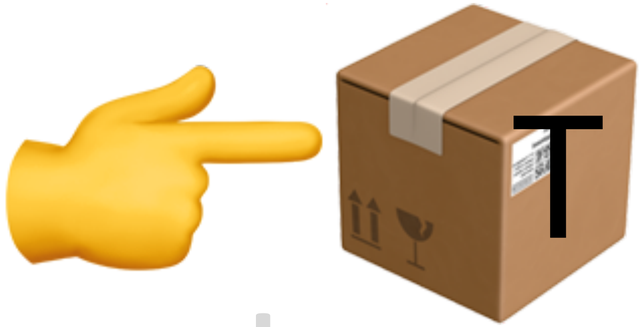
designating

managing

variant<...>

type erasure

requires heap allocation or s-o-o



value

polymorphic

designating

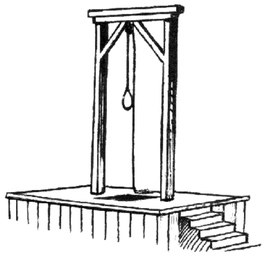
managing

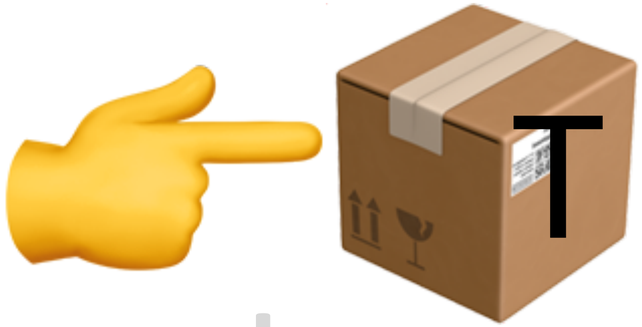
regular (~~T&~~)

copyable (~~T&~~)

improper

potentially





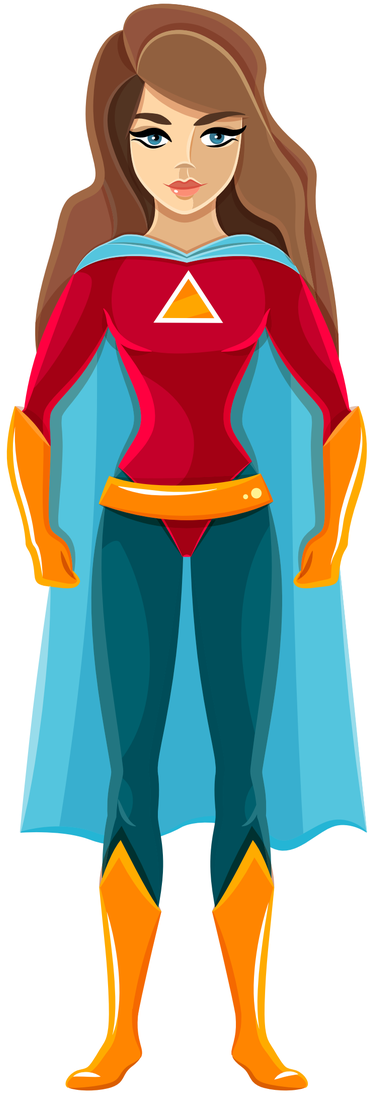
value

polymorphic

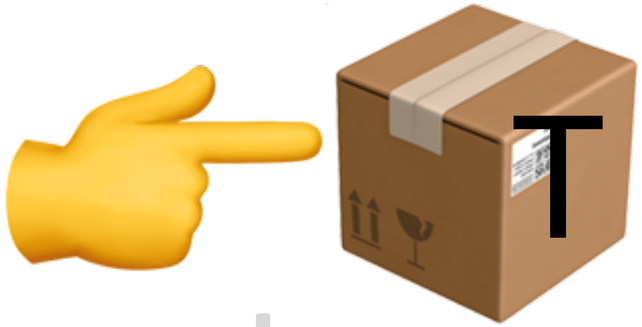
designating

managing

T*



Background vector created by macrovector - www.freepik.com

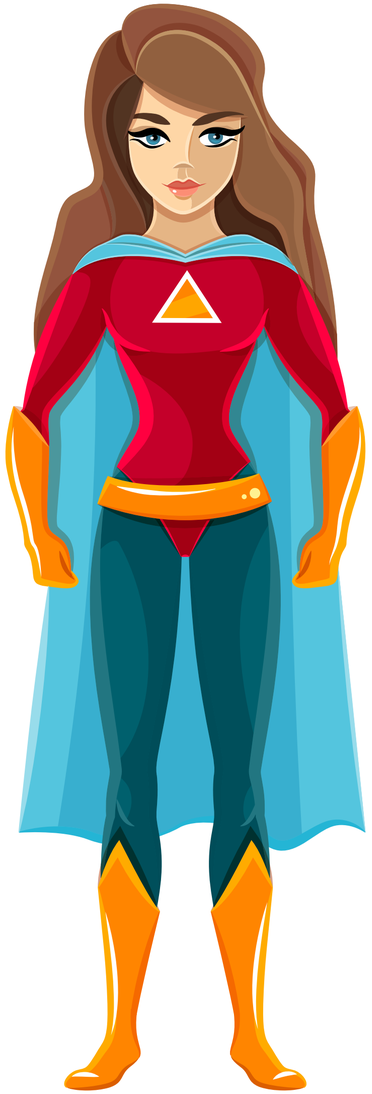


value

polymorphic

designating

managing



value instead

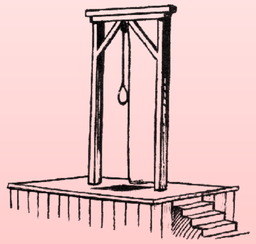
only local scope

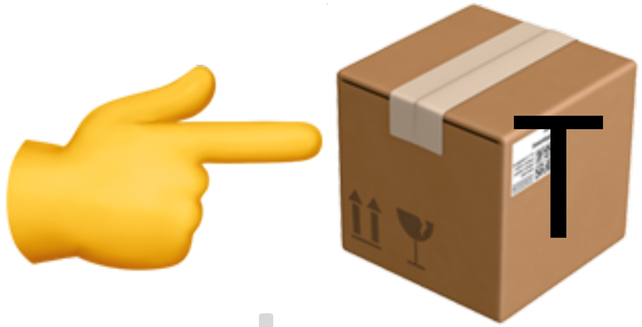
manage

encapsulate

Background vector created by macrovector - www.freepik.com

Mitigation dangling for single object designators (aka T*) 👉📦

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&	
		rebind	reference_wrapper<T>	to get assignability with a reference memnber
	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&>	



value

polymorphic

designating

managing

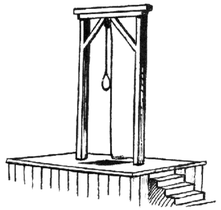
as:

member

base

parameter

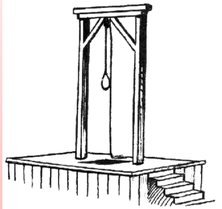
return



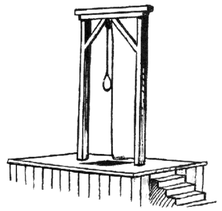
```
struct Dangle {
    Dangle& get() {
        ++val;
        return *this;
    }
    ~Dangle(){val=1;}
    int val{42};
};
```

```
void demoDangle(){
  Dangle& dang{Dangle{}.get()};
  int res{dang.val}; // UB
  ASSERT_EQUAL(1,res);

  res = Dangle{}.get().val; // OK
  ASSERT_EQUAL(43,res);
}
```



```
void demoVectorDangle(){
  auto it=std::vector<int>{1,2,3,4}.begin();
  ASSERT_EQUAL(2,*++it); // UB!
}
```



.get() breaks encapsulation

Returning Designating Types from Members is especially dangerous

```
struct NoTempDangle {
  NoTempDangle& get() & {
          ++val;
          return *this;
      }
      ~NoTempDangle(){val=1;}
      int val{42};
};
```

```
void demoNoTempDangle(){
  int res = NoTempDangle{}.get().val; // error
}
```

```
../src/Test.cpp:41:12: error: 'this' argument to member function 'get'
is an rvalue, but function has non-const lvalue ref-qualifier
        int res = NoTempDangle{}.get().val; // does not compile
                  ^
```

```
template <class _Tp, class _Allocator>
inline _LIBCPP_INLINE_VISIBILITY
typename vector<_Tp, _Allocator>::iterator
vector<_Tp, _Allocator>::begin() _NOEXCEPT
{
    return __make_iter(this->__begin_);
}

template <class _Tp, class _Allocator>
inline _LIBCPP_INLINE_VISIBILITY
typename vector<_Tp, _Allocator>::const_iterator
vector<_Tp, _Allocator>::begin() const _NOEXCEPT
{
    return __make_iter(this->__begin_);
}
```

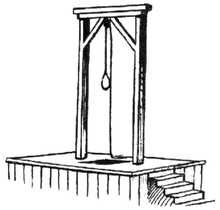
&

&

unfixable legacy?

Ref-qualify member functions returning a *Designating* Type referring (part of) *this

It can be worse.... Even the defaults are dangerous....



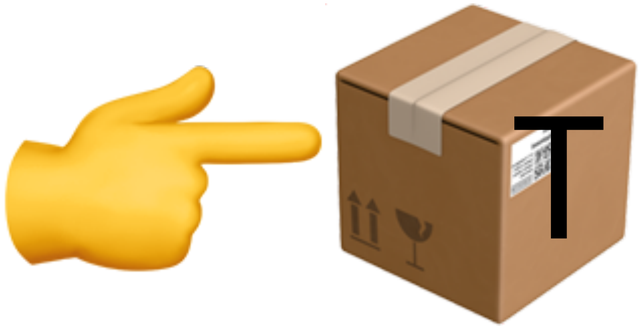
```
struct AssignDangle{
  int i{};
  AssignDangle(int j):i{j}{}
  ~AssignDangle(){ i=42; }
  AssignDangle& operator=(AssignDangle const &) & = default;
};
```

```
void demoAssignDangle(){
  auto & ad = (AssignDangle{1} = AssignDangle{2});
  ASSERT_EQUAL(2,ad.i); // UB, fails, 42!
}
```

```
../src/Test.cpp:64:31: error: no viable overloaded '='
        auto & ad = (AssignDangle{1} = AssignDangle{2});
                     ~~~~~~~~~~~~~~~ ^ ~~~~~~~~~~~~~~~
../src/Test.cpp:60:16: note: candidate function not viable: no known conversion from 'fixed::AssignDangle' to
'fixed::AssignDangle' for object argument
        AssignDangle& operator=(AssignDangle const &) & = default;
```

```
class Good {
public:
  Good() = default;
  ~Good() = default;
  Good(const Good &other) = default;
  Good(Good &&other) noexcept = default;
  Good& operator=(const Good &other) & = default;
  Good& operator=(Good &&other) & noexcept = default;
};
```

Is that better?



```
class Ptr {
  int *pi{};
public:
  Ptr() = default;
  explicit Ptr(int *pp):pi{pp}{}
  int const *get() const { return pi; }
};
```

Designation is contagious

it depends!

value

polymorphic

designating

managing

as:

member ❓

base ❌

parameter 👉

return ❓

value

polymorphic

designating

managing

non-copyable ?

unique ?

proper ?

regular ?

value

polymorphic

designating

managing

scoped ✅ {}

unique

shared ❓

value ✅ 💰

Scoped Manager 

```
class Scoped {
  Resource resource;
public:
  Scoped(); // acquire resource
  ~Scoped();// release resource
  Scoped& operator=(Scoped &&other) = delete;
};
```

DesDeMovA

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

Unique Manager

```
class Unique {
  std::optional<Resource> resource;
  void release() noexcept;// release resource
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();
}
```

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

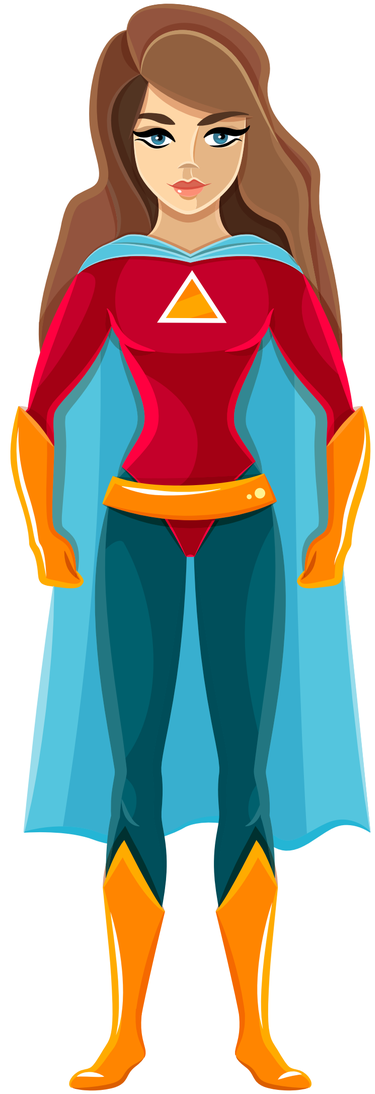
```
Unique& Unique::operator=(Unique &&other) & noexcept {
  if (this != &other) {
    this->release();
    std::swap(this->resource, other.resource);
  }
  return *this;
}
```

Shared Manager

use
shared_ptr<resource>
and concurrency control
if at all

Manager as Value

```
class MValue {
public:
  MValue() = default;
  ~MValue();
  MValue(MValue &&other) noexcept ;
  MValue& operator=(MValue &&other) & noexcept;
  MValue(const MValue &other);
  MValue& operator=(const MValue &other) &;
};
```



Hard, but rewarding!

Background vector created by macrovector - www.freepik.com

A quick quiz on class natures:

std::vector<T> is _____

std::scoped_lock is _____

std::unique_ptr<T> is _____

std::shared_ptr<T const> is _____

std::thread is _____

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Be aware there can still be strange creatures!

As Member:

value ✅ 💰

polymorphic 👔 👉 💰

designating 🦠 👔

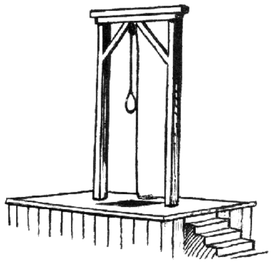
managing {} 🏝 💰

	Some constructor	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Value	yes	none / =default	defaulted	defaulted	defaulted	defaulted	defaulted
Designating	yes	none/ =default	defaulted	defaulted/ deleted	defaulted/ deleted	defaulted/ deleted	defaulted/ deleted
Manager **Scoped**	typical	none	implemented	deleted	deleted	deleted	=delete 🙋🏼
Manager **Unique**	typical	defined / =default	implemented	deleted	deleted	implemented	implemented
Manager **Value**	yes	defined / =default	implemented	implemented	implemented	implemented	implemented
OO - Base	protected	protected	virtual =default	deleted	deleted	deleted	=delete 🙋🏼

Influence of Member Variable Types on Special Member Functions

Member Varible Nature	Some constructor	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Value	none/typical	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Manager Scope [2]	typical	none	defaulted	deleted	deleted	deleted	deleted
Manager Unique [2]	typical	defined / =default	defaulted	deleted	deleted	defaulted	defaulted
Designating [2,3]	typical	defined / =default	defaulted	defaulted	defaulted	defaulted	defaulted
T& [2]	yes	=delete	defaulted	defaulted	=delete [1]	defaulted	=delete [1]

- 1 - remedy through using std::reference_wrapper<T> instead
- 2 - "contagious": your class becomes the same without further means
- 3 - Regular Designating Members make using your class type dangerous



- Be conscious of your class' nature 
- Select your special member functions according to their canonical form
- 💰 rules
- Be aware of designating types that can come in disguise
- Minimize surprising class natures or encapsulate them

At SIX, we have successfully integrated Cevelop into the development environment of our trading applications area, in order to develop a C++ replacement for an existing Java application. We apply regular Cevelop updates, make use of the integrated Cute testing framework and have utilized several plugins with their convenient 'quickfixes'. As an example, we are using the Ctyle plugin to aid us in writing modern C++ source code.

Cevelop is s a coherent overall package that greatly facilitates our development.

Cevelop

Your C++ deserves it

Download IDE at:
www.cevelop.com

Questions?

peter.sommerlad@hsr.ch
@PeterSommerlad

"By the way, thanks for the great piece of software! This is by far the best free IDE for C++ so far after trying all the free C/C++ IDE."
motowizlee on github 27.04.2017