Cevelop

Your C++ deserves it

# Sane and Safe C++ Class Types

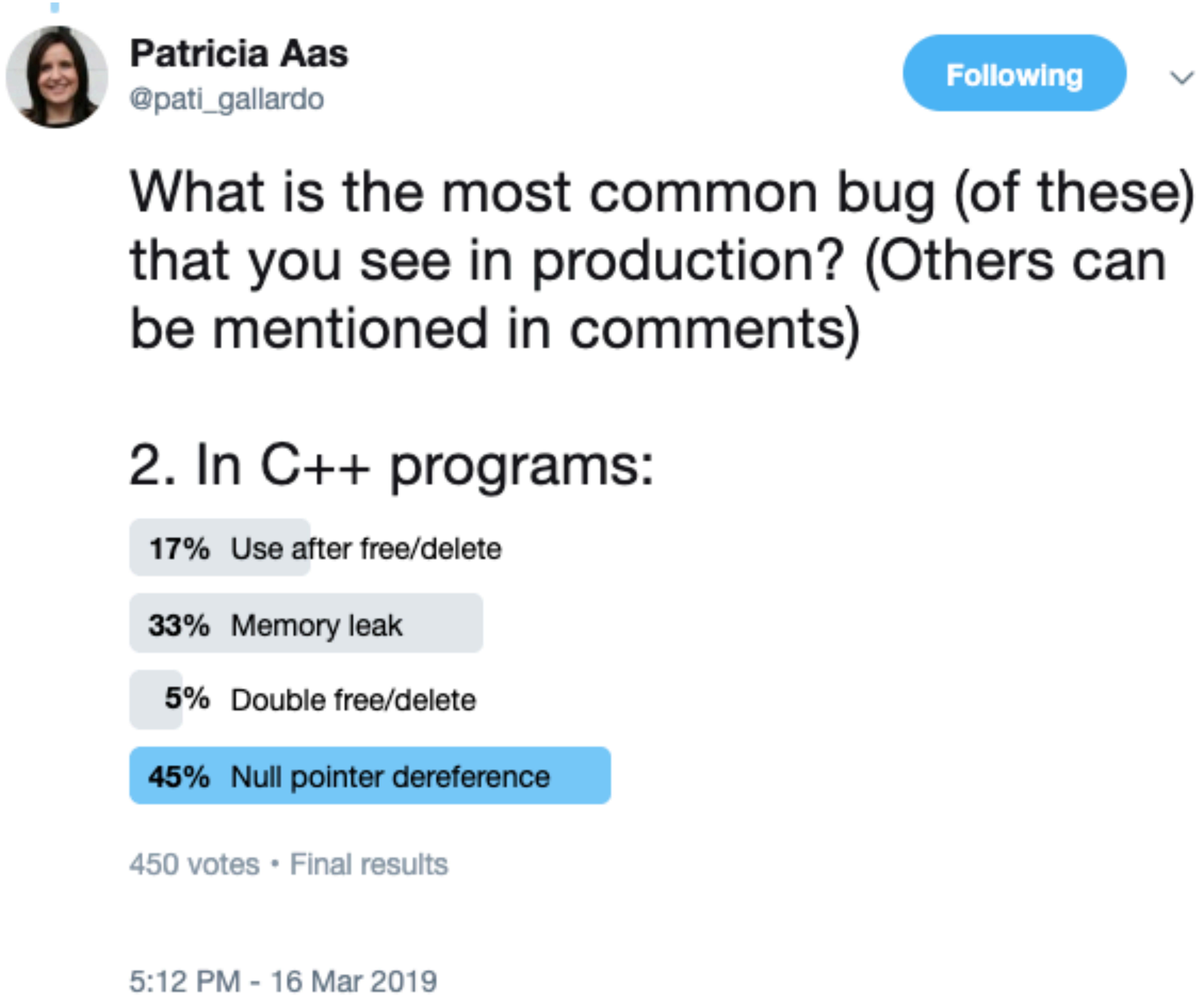
Prof. Peter Sommerlad
Director of IFS
ACCU April 2019

@PeterSommerlad
peter.cpp@sommerlad.ch

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

# Typical C++ problems ?



**Patricia Aas**
@pati_gallardo

What is the most common bug (of these) that you see in production? (Others can be mentioned in comments)

2. In C++ programs:

17% Use after free/delete

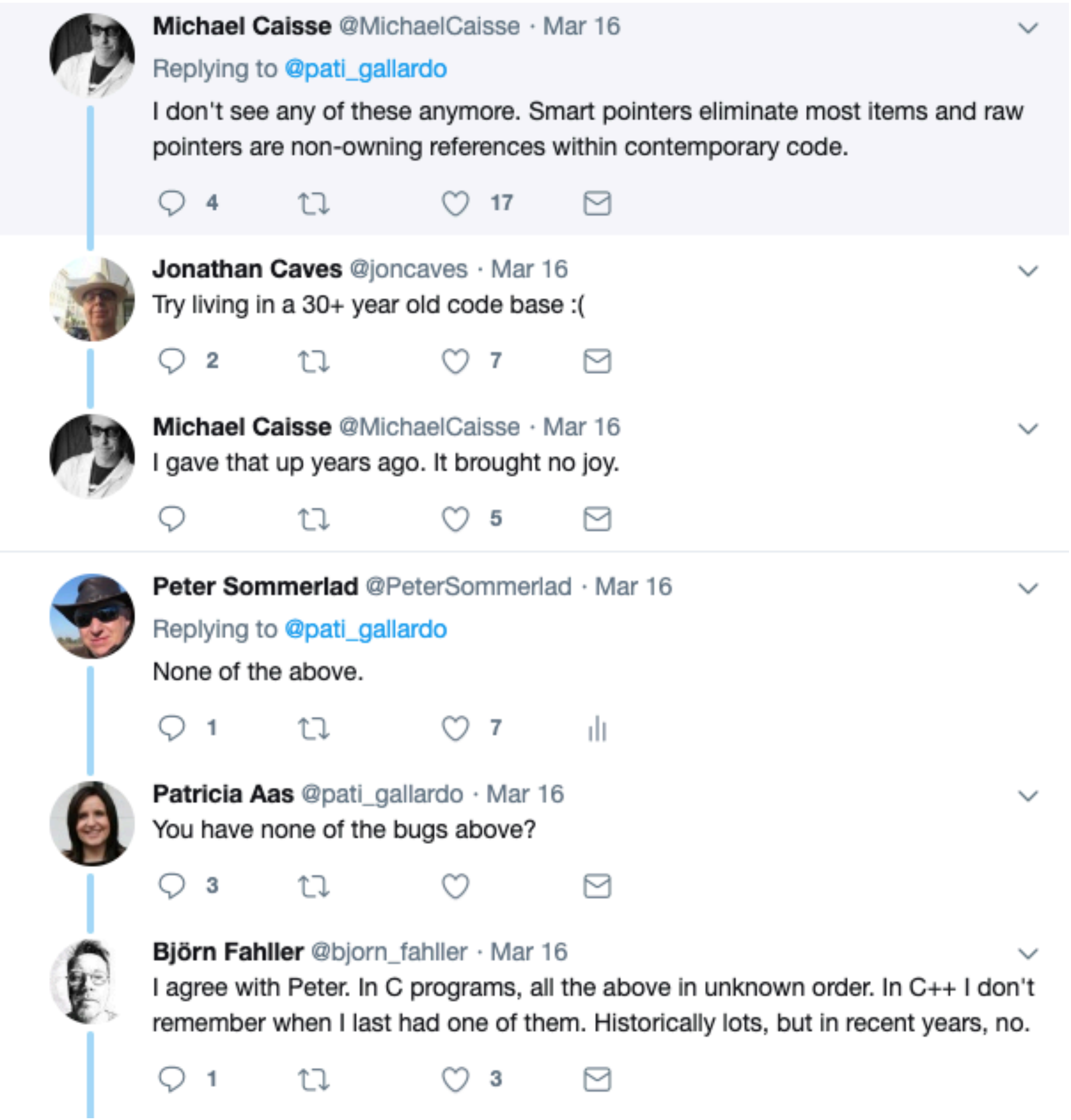
33% Memory leak

5% Double free/delete

45% Null pointer dereference

450 votes • Final results

5:12 PM - 16 Mar 2019



**Michael Caisse** @MichaelCaisse · Mar 16
Replying to @pati_gallardo
I don't see any of these anymore. Smart pointers eliminate most items and raw pointers are non-owning references within contemporary code.

**Jonathan Caves** @joncaves · Mar 16
Try living in a 30+ year old code base :(

**Michael Caisse** @MichaelCaisse · Mar 16
I gave that up years ago. It brought no joy.

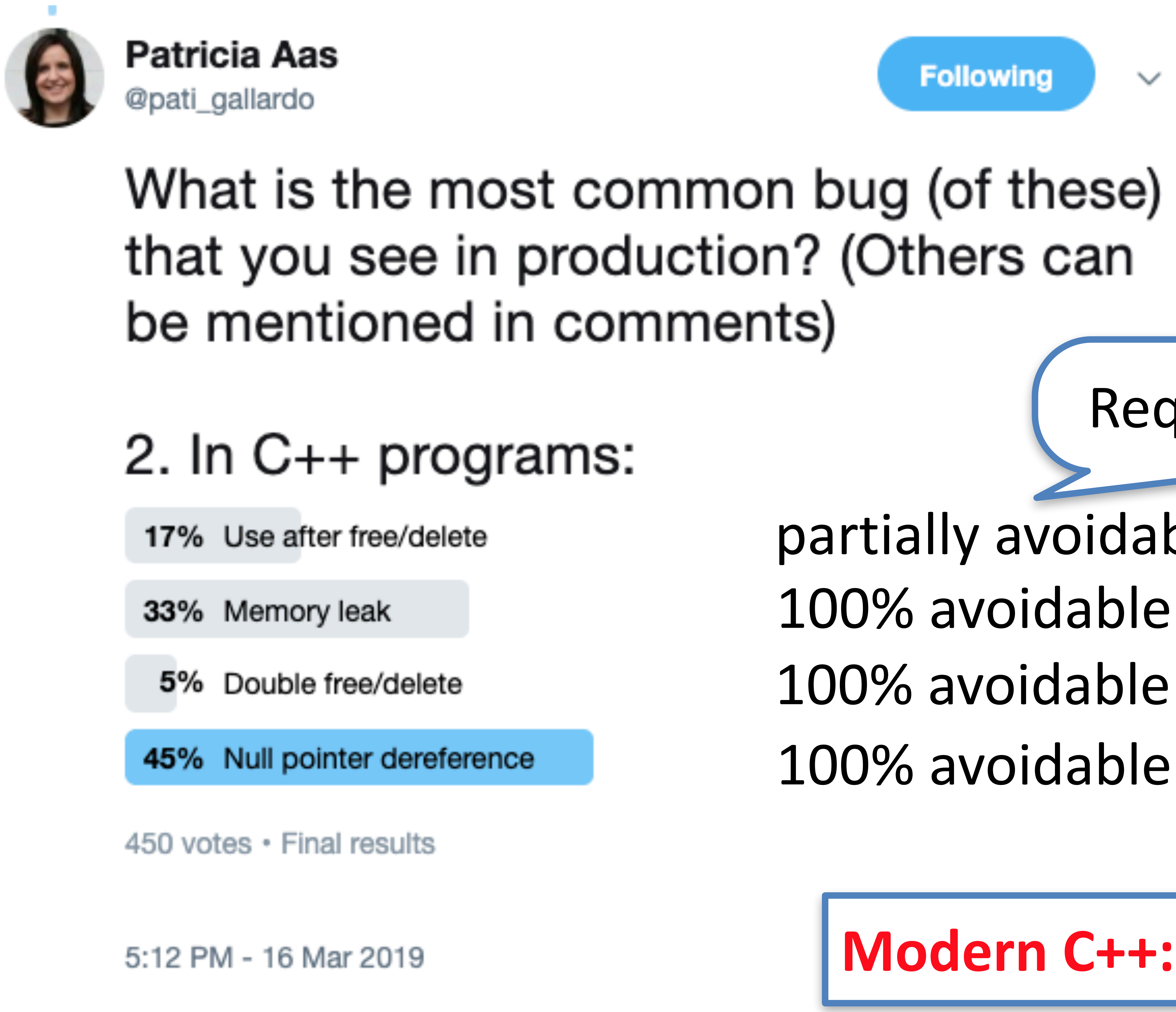
**Peter Sommerlad** @PeterSommerlad · Mar 16
Replying to @pati_gallardo
None of the above.

**Patricia Aas** @pati_gallardo · Mar 16
You have none of the bugs above?

**Björn Fahller** @bjorn_fahller · Mar 16
I agree with Peter. In C programs, all the above in unknown order. In C++ I don't remember when I last had one of them. Historically lots, but in recent years, no.

© Peter Sommerlad

# C bugs are 100% avoidable in modern C++, except for dangling



Patricia Aas
@pati_gallardo

What is the most common bug (of these) that you see in production? (Others can be mentioned in comments)

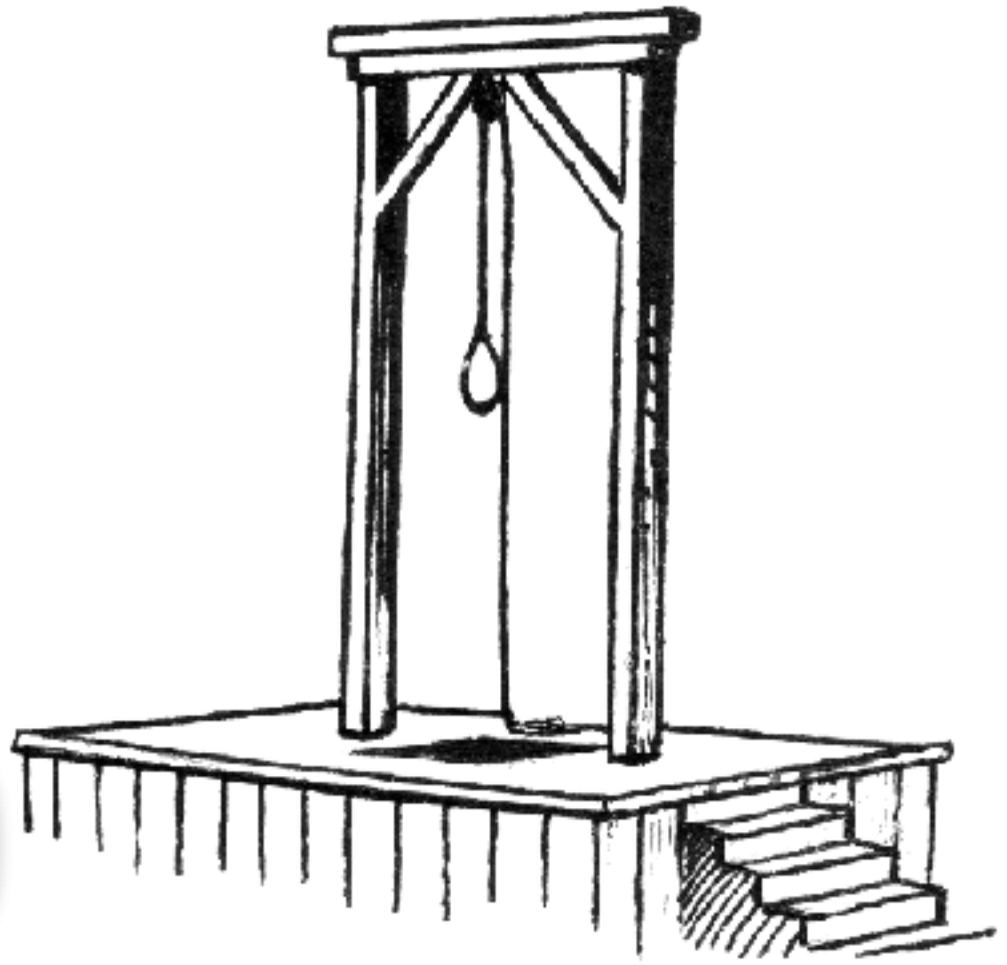
2. In C++ programs:

| | |
|---|---|
| 17% | Use after free/delete |
| 33% | Memory leak |
| 5% | Double free/delete |
| 45% | Null pointer dereference |

450 votes • Final results

5:12 PM - 16 Mar 2019

Requires Discipline!



partially avoidable: values instead of pointers, references, views

100% avoidable: unique_ptr, containers, or values

100% avoidable: unique_ptr, containers, or values

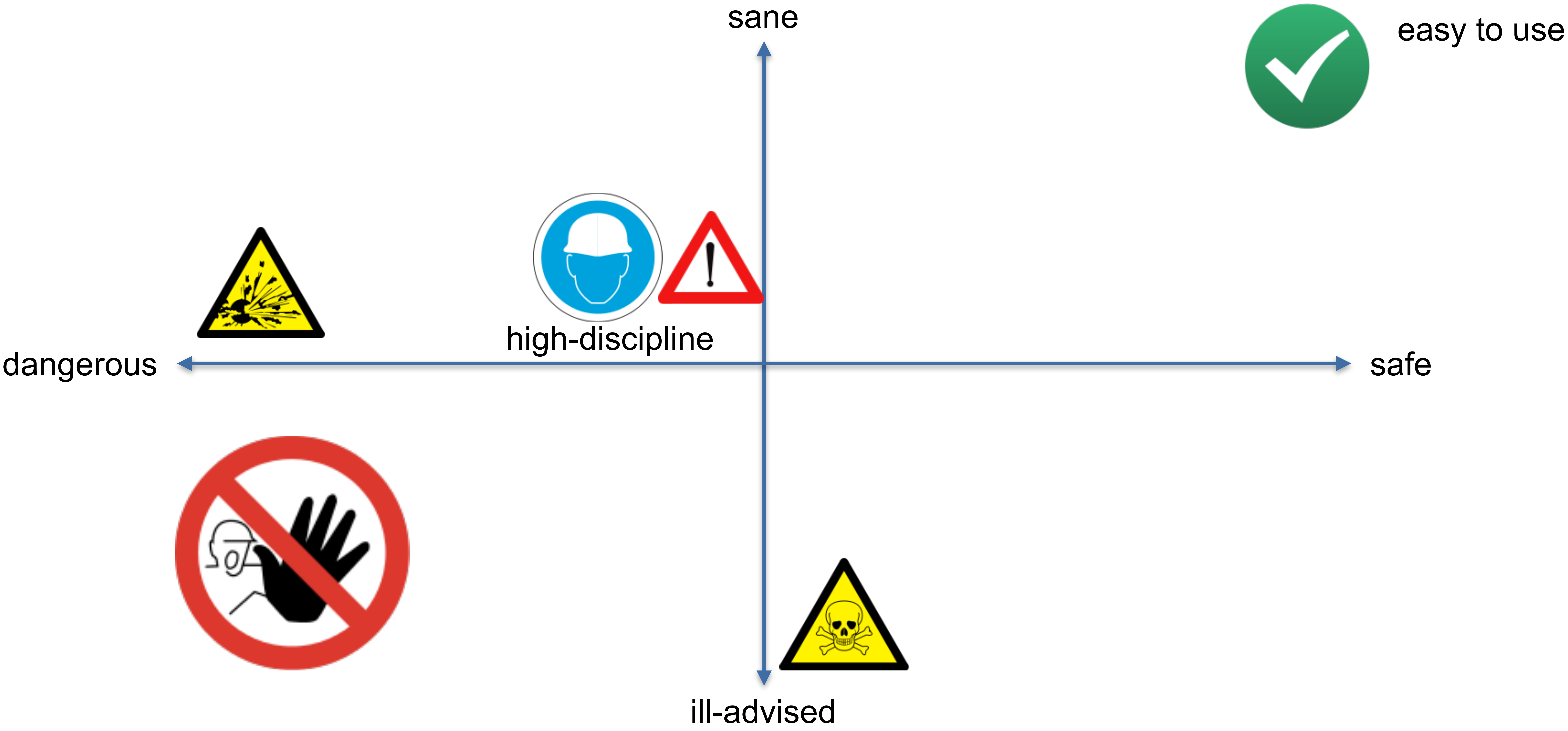
100% avoidable: values, references, checks (gsl::not_null)

**Modern C++: NO PLAIN POINTERS or C-ARRAYS**

except tightly encapsulated

© Peter Sommerlad

# Dimensions Safety and Sanity...



© Peter Sommerlad

# Safe and sane user-defined class types

- **Value Types**
- **Empty Types**
- **Managing Types (different flavors)**
- **OO-polymorphic-hierarchy Types**

- **semi-sane: "potentially dangling object types" aka "pointing types"**

© Peter Sommerlad

# Values

"When in doubt, do as the ints do!"
 -- Scott Meyers

"But may be not always..."
 -- Peter Sommerlad

© Peter Sommerlad

# Value Types - Regular Types

C++ standard containers assume (semi-)regular types as template arguments for elements. They might work with non-default constructible or move-only types but with limited functionality.

👍 **Just Works™**

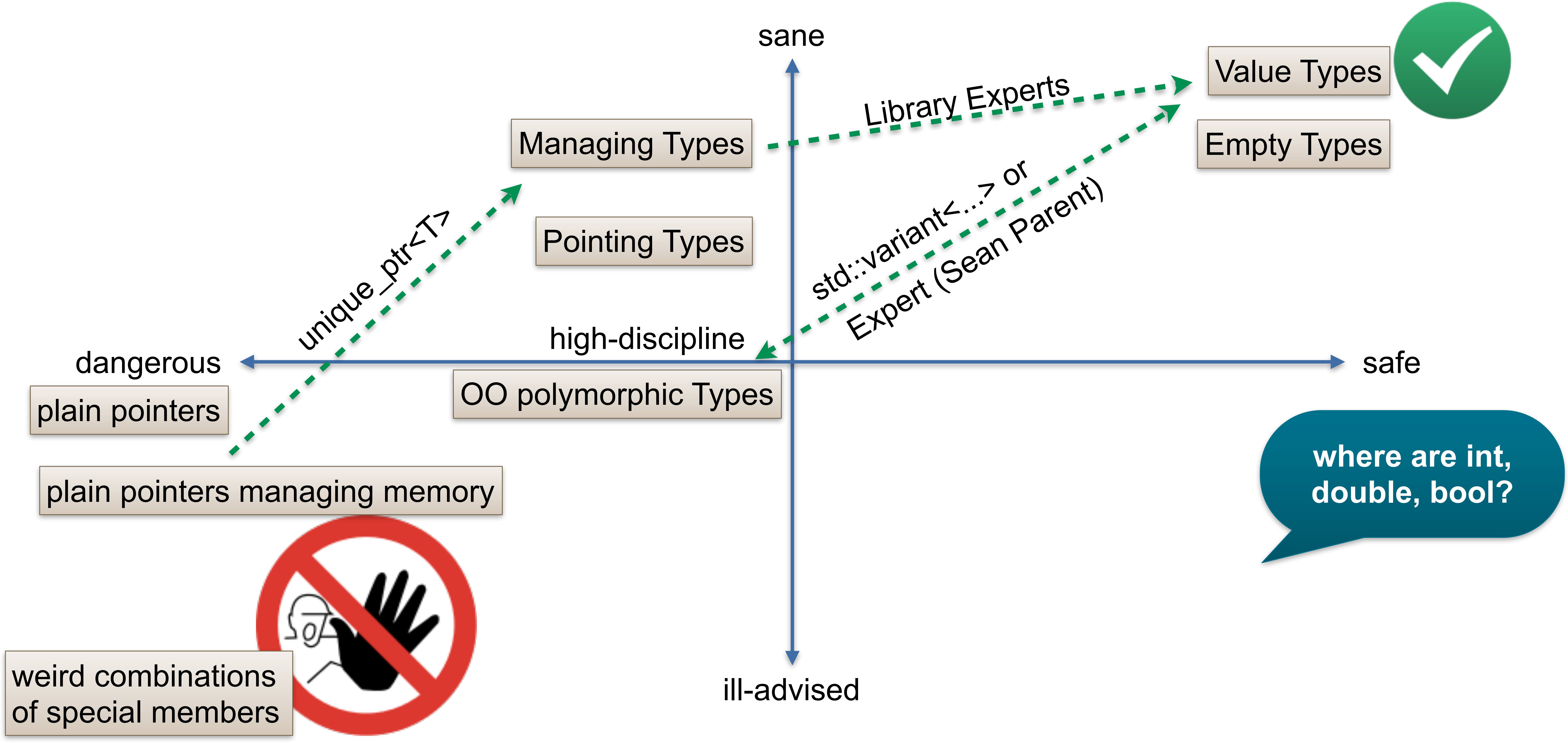
- **Properties of types satisfying concept Regular<T>**
  - EqualityComparable (==, !=)
  - DefaultConstructible T{}
  - Copyable T(T const&), T& operator=(T const&)&,
  - Movable T(T&&), T& operator=(T&&)&, is_object_v
  - Swappable swap(T&,T&)
  - Assignable t1 = t2
  - MoveConstructible T(T&&)

Sometimes Ordering is also required std::less<T> should work, usually by defining bool operator<(T,T)

If comparison works, it should be consistent!
C++20 will make that **differently**, through the "spaceship" operator<=>

© Peter Sommerlad

# Value (class) Types are Sane and Safe!



where are int, double, bool?

© Peter Sommerlad

# Are primitive language types "sane" and "safe"?

- **Safety: int, char, bool, double are Regular value types, OK**
  - copying, equality is given
- **BUT:**

```
void InsaneBool() {
  using namespace std::string_literals;
  auto const i { 41 };
  bool const throdd = i % 3;
  auto const theanswer= (throdd & (i+1) ) ? "yes"s : "no"s;
  ASSERT_EQUAL("",theanswer);
}
```

What makes the test run?

© Peter Sommerlad

# Are language types "sane" and "safe"?

- **Safety: int, char, bool, double are Regular value types, OK**
  - copying, equality is given
- **BUT:**

```
void InterestingSetDouble(){
  std::vector v{0.0,0.01,0.2,3.0};
  std::set<double> s{};
  for (auto x:v){
    for (auto y:v)
      s.insert(x/y);
  }
  ASSERT_EQUAL(v.size()*v.size()-v.size()+1,s.size()); // really?
}
```

What is the size?

© Peter Sommerlad

# Are library types "sane" and "safe"?

- **Safety: containers are Regular value types, if their elements and other template arguments are.**
  - copying, equality is given
- **BUT: they still use built-in types resulting in interesting behavior**

```
void printBackwards(std::ostream &out, std::vector<int> const &v){
  for(auto i=v.size() - 1; i >= 0; --i)
    out << v[i] << " ";
}
```

**Can you spot the bug!**

# Some of the Problems with primitive built-in types

- **Integral promotion (inherited from C)**
  - with very interesting rules no one can remember correctly, including bool and char as integer types
  - signed - unsigned mixtures in arithmetic
  - silent wrapping vs. undefined behavior on overflow, vs. signaling of overflow (want the carry bit!)
- **Automatic (numeric) conversions**
  - integers <-> floating points <-> bool
  - and that complicated with types with non-explicit constructors and conversion operators
- **Special values for floating point numbers**
  - +Inf, -Inf, NaN (often forgotten)

warnings often silenced with arbitrary casts

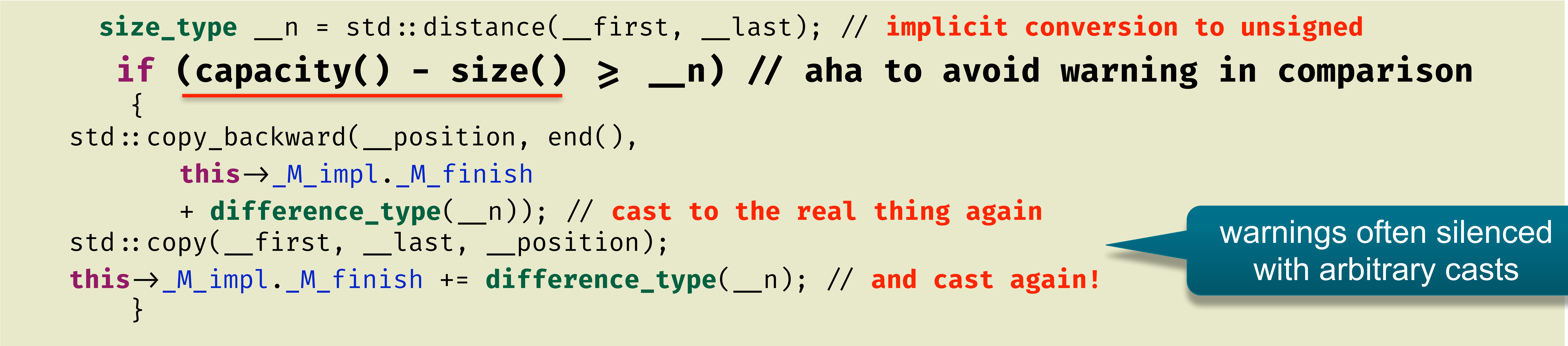
Do not make your class types implicitly convert!

Make comparison strict weak order or stronger!

© Peter Sommerlad

# Further problems of using primitive types in C++

- **Consciously wrap primitive, or built-in types into types with meaning to the application**
  - fluximate(int,int,int) is hard to call correctly! fluximate(3,2,1) or fluximate(1,2,3)
    - BTW: Named Parameters are only curing a symptom (IMHO in the wrong way)!
  - C++ can do so without (significant) run-time overhead
- **Standard library is guilty of using built-ins as type aliases where they do not fit nicely**
  - size_t, size_type --> count elements = natural numbers including 0 - **absolute value**

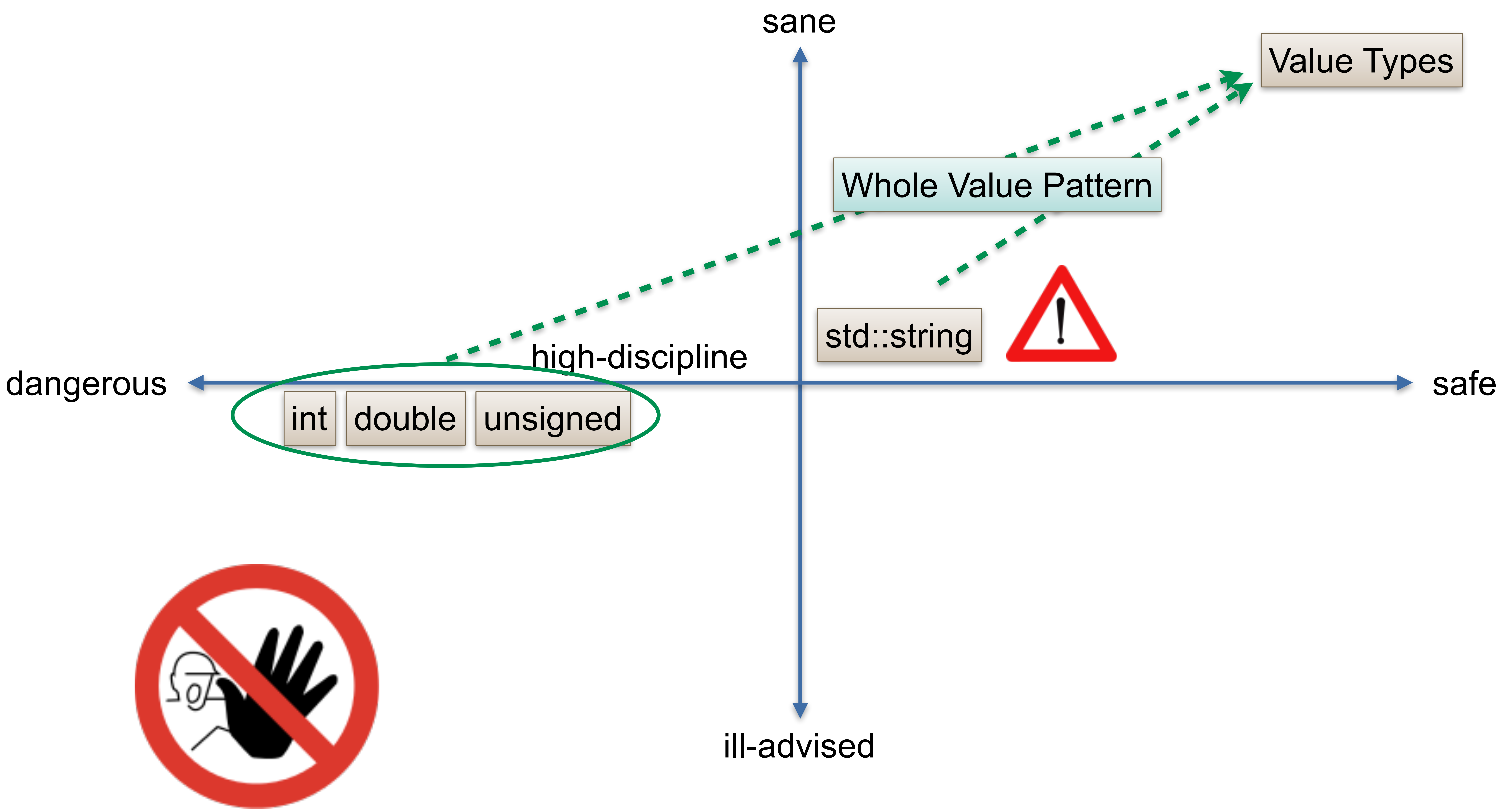


```
      size_type __n = std::distance(__first, __last); // implicit conversion to unsigned
       if (capacity() - size() ≥ __n) // aha to avoid warning in comparison
        {
    std::copy_backward(__position, end(),
           this→_M_impl._M_finish
           + difference_type(__n)); // cast to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

warnings often silenced with arbitrary casts

© Peter Sommerlad

# Dimensions Safety and Sanity…



© Peter Sommerlad

# Whole Value Pattern - motivation

```
check_counters(0,1);// which is which?
```

- Parameters can be confusing, when multiple parameters of the same type occur.
- Names can help, but...
- Some time ago, an IFS assistant searched for a bug, where two arguments were in the wrong order

```
void check_counters(size_t waits, size_t notifies);
```

- Type aliases as in the standard library are no solution:

```
using WaitCounter=size_t;
using NotifyCounter=size_t;
void check_counters(WaitCounter w, NotifyCounter n);
```

- Need: "Strong" Type Aliases - each role/usage gets its own type that is not a primitive type

© Peter Sommerlad

# Whole Value Pattern (Ward Cunningham - CHECKS Pattern Language)

- *When parameterizing or otherwise quantifying a business (domain) model there remains an overwhelming desire to express these parameters in the most fundamental units of computation.*

like C

  - *Not only is this no longer necessary (it was standard practice in languages with weak or no abstraction), it actually interferes with smooth and proper communication between the parts of your program and with its users.*
  - *Because bits, strings and numbers can be used to **represent almost anything**, any one in isolation **means** almost **nothing**.*

- **Therefore:**
- *Construct specialized values to quantify your domain model and use these values as the **arguments** of their messages and as the units of input and output.*

Value Types

  - *Make sure these objects capture the whole quantity with all its implications beyond merely magnitude, but, keep them independent of any particular domain.*

functions, operators

  - *Include format converters in your user-interface that can correctly and reliably construct these objects on input and print them on output.*

constructors, I/O

  - ***Do not expect your domain model to handle string or numeric representations of the same information.***

no implicit conversions

© Peter Sommerlad

# What does that mean?

```
check_counters(0,1);// which is which?
```

**Whenever you have a function taking multiple arguments of the same type,**

**it will be called wrongly!**

© Peter Sommerlad

# Whole Value Pattern in the most simple way: just define a struct type

```
check_counters(Wait{0},Notify{2});
```

Aggregate Initialization: structtype{members}

- **Documents which counter has which role at call site (note: no implicit constructors!)**
- **Overloading is possible to allow more flexibility (but not necessarily recommended)**

```
void check_counters(Wait w, Notify n);
```

- **Define a struct/class wrapping the simple type (with required operators):**

```
struct Wait {
  size_t count{};
}; // minimal version
```

**The simplest strong type version**

```
void operator++(Wait &w){ // retrofit increment for use case
  w.count++;
}
```

© Peter Sommerlad

# Well, we have two counters.. Avoiding Duplication

- **Common attempt: Extract Base Class --> Not that simple...**

```
struct CounterBase{
  size_t count;
  void operator++(){ // what to return?
    ++count;
  }
  bool operator==(CounterBase const &other)const{
    return count==other.count;
  }
};
struct WaitB:CounterBase{};
struct NotifiesB:CounterBase{};
```

DANGER

No more separation

DANGER

delete via base pointer

```
void CompareWaitsWithNotifies() {
  WaitB waits{5};
  ASSERT_EQUAL(NotifiesB{5},waits);
}
```

© Peter Sommerlad

# CRTP: Curiously Recurring Template Parameter Pattern

- **Extract Templated Base Class:**

```
template <typename TAG>
struct Counter{
  size_t count{};
  bool operator==(Counter const &other) const {
    return count == other.count;
  }
  Counter& operator++(){
    ++count;
    return *this;
  }
};
struct Wait:Counter<Wait> {
};
struct Notify:Counter<Notify> {
};
```

```
void CompareWaitsWithNotifiesCRTP() {
  Wait waits{5};
  ASSERT_EQUAL(Notify{5},waits);
}
```

Does not compile!

```
../src/Test.cpp:9:7: note:   no known
conversion for argument 1 from 'const
Wait' to 'const Counter<Notify>&'
```

DANGER

delete via
base pointer

© Peter Sommerlad

# Should I make a (whole) value type default-constructible? T()=default;

- **Yes, whenever there is a natural default or neutral value in your type's domain**
  - int{} == 0
  - Be aware that the neutral value can depend on the major operation: int{} is not good for multiplication
- **May be, when initialization can be conditional and you need to define a variable first**
  - consider learning how to use ?: operator or an in-place called lambda, requires assignability otherwise
- **No, when there is not natural default value**
  - PokerCard (2-10, J, Q, K, Ace of ♤♧♡♢) What should be the default? - no default constructor!
- **No, when the type's invariant requires a reasonable initialization**
  - e.g., class CryptographicKey --> to be useful needs real key data

© Peter Sommerlad

# Units beware: relative vs. absolute - often misunderstood or easily misapplied

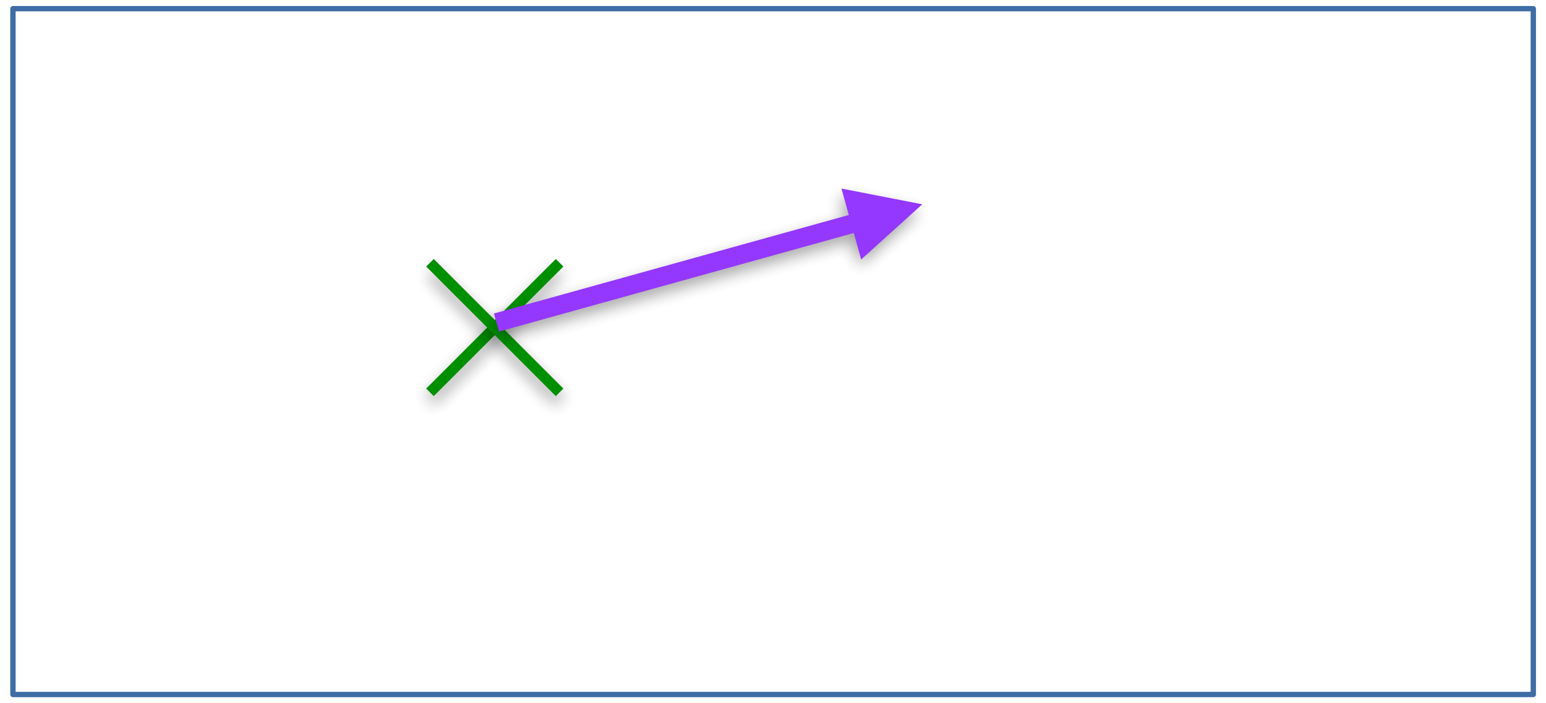
```
      size_type __n = std::distance(__first, __last); // implicit conversion to unsigned
       if (capacity() - size() ≥ __n) // aha to avoid warning in comparison
        {
   std::copy_backward(__position, end(),
           this→_M_impl._M_finish
           + difference_type(__n)); // cast to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

relative

relative

wrong result type!

- **<chrono> is a good example to follow:**
  - time_point and duration: tp1 - tp2 -> duration, tp + d -> time_point, **tp+tp -> nonsense**, d1 + d2 -> duration
- **position vs. direction**
  - Vec3d/Vec3 and similar are problematic, because identical representation is used for both roles
  - location and displacement
- **generic units must make this distinction**
  - easily forgotten in dimensional analysis



"affine spaces"

# More generic "Strong" Types

- see video presentations and libraries by
- Björn Fahller (ACCU2018)
- Jonathan Boccara
- Jonathan Müller
- Me: PSST - Peter's simple strong typing
  - uses aggregates and CRTP mix-ins (work in progress)

```
struct WaitC:strong<unsigned,WaitC>
            ,ops<WaitC,Eq,Inc,Out>{};
static_assert(sizeof(unsigned)==sizeof(WaitC));

void testWaitCounter(){
  WaitC c{};
  WaitC const one{1};
  ASSERT_EQUAL(WaitC{0},c);
  ASSERT_EQUAL(one,++c);
  ASSERT_EQUAL(one,c++);
  ASSERT_EQUAL(2,c.get());
}
```

CRTP

EBO

- IMHO, "Strong Typing" frameworks/infrastructure are often too generic.
- Aggregate types are OK -> Rule of Zero, No automatic conversion, unless specified!
  - If there is no invariant to be ensured, ie., all member-type values are valid
  - C++17 allows operations to be CRTP-mixed-in without space overhead, if first base contains actual value

`delete via base pointer`

Thanks Loïc Joly

## Empty Classes - useful?

"Oh you don't get something for nothing"
-- Rush

"Something for Nothting" -- Kevlin Henney, 1999

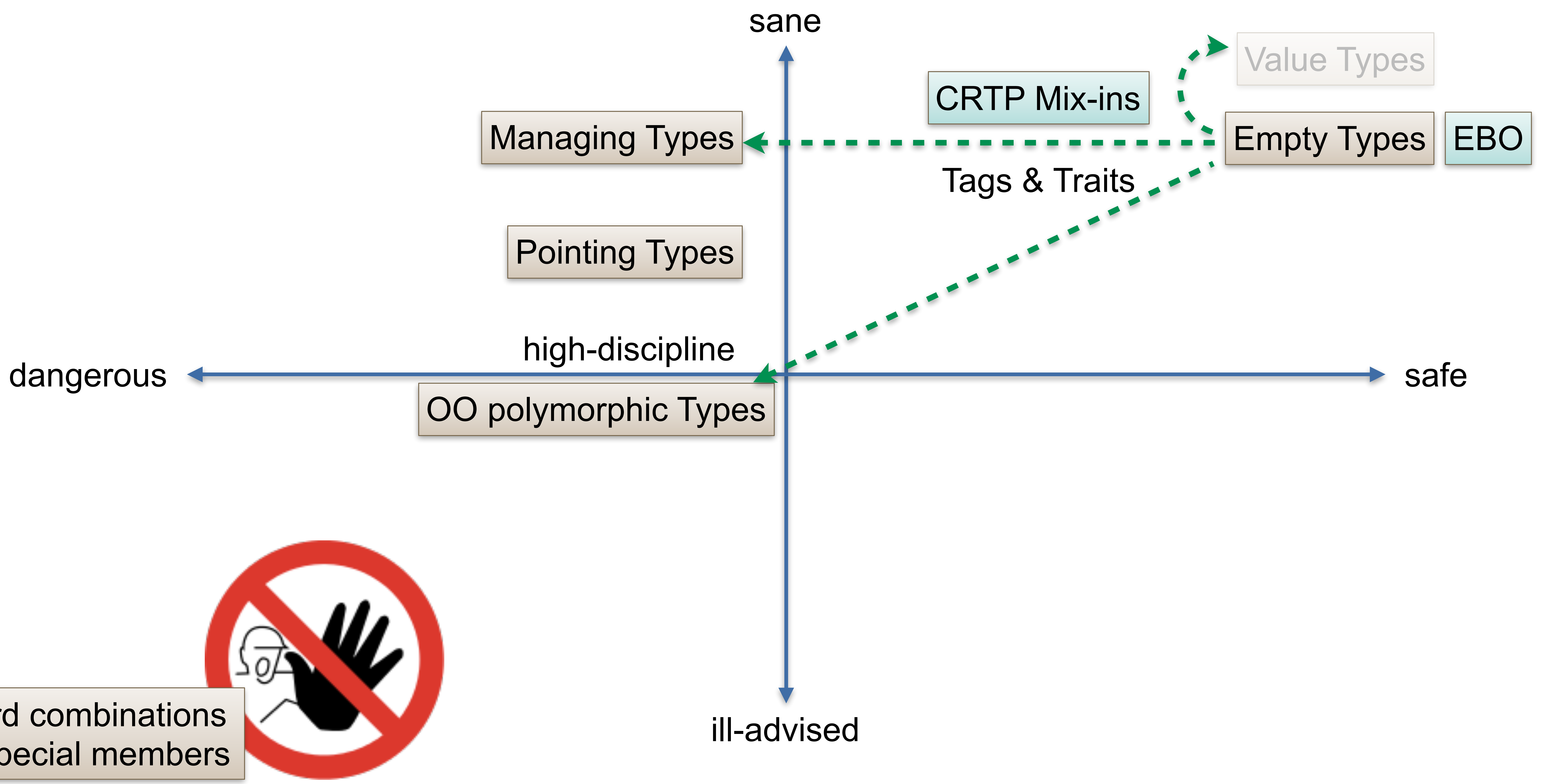
With a C++ Empty Class
you get something for nothing!

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

# Dimensions Safety and Sanity…



# Tag Types: Overload selection - sometimes with universally usable constants

- **Iterator Tags**
  - `input_iterator_tag,`
    `output_iterator_tag,`
    `forward_iterator_tag,`
    `bidirectional_iterator_tag,`
    `random_access_iterator_tag`
- **in place marker: `in_place_t`**
  - `std::in_place` global value

```
template< class... Args >
constexpr explicit
optional( std::in_place_t, Args&&... args );



// calls std::string( size_type count, CharT ch ) constructor
std::optional<std::string> o5(std::in_place, 3, 'A');
```

```
template< class BDIter >
void alg(BDIter, BDIter, std::bidirectional_iterator_tag)
{
    std::cout << "alg() called for bidirectional iterator\n";
}
template <class RAIter>
void alg(RAIter, RAIter, std::random_access_iterator_tag)
{
    std::cout << "alg() called for random-access iterator\n";
}
template< class Iter >
void alg(Iter first, Iter last)
{
    alg(first, last,
        typename std::iterator_traits<Iter>::iterator_category());
}
int main()
{
    std::vector<int> v;
    alg(v.begin(), v.end());

    std::list<int> l;
    alg(l.begin(), l.end());

//    std::istreambuf_iterator<char> i1(std::cin), i2;
//    alg(i1, i2); // compile error: no matching function for call
}
```

- `nullptr_t` and `nullptr` are similar but built-in

© Peter Sommerlad

# Traits - compile-time-meta-programming <type_traits> <ratio>- values as types

- **represent values as types**
    - integral_constant<T,T v>
        - true_type, false_type
    - ratio<5,3>
    - integer_sequence<T, T...vs>
- **What for?**
- **SFINAE**
    - template specialization selection
    - overload selection
- **Periods/scale in duration (ratio)**
- **tuple element access (integer_sequence)**

```
template<class T, T v>
struct integral_constant {
    using value_type=T;
    static constexpr value_type value = v;
    using type=integral_constant; // injected-class-name
    constexpr operator value_type() const noexcept {
        return value; }
    constexpr value_type operator()() const noexcept {
        return value; }
};
using true_type=integral_constant<bool,true>;

static_assert(integral_constant<bool,true>::value,"");
static_assert(true_type::value,"member access");
static_assert(true_type{},"auto-conversion");
static_assert(true_type{}(),"call operator");
static_assert(std::is_same_v<true_type, true_type::type>,
     "type meta");
```

© Peter Sommerlad

# Traits - compile-time-meta-programming <type_traits> - type properties

- **determine type properties ..._v**
  - constexpr bool variable template
- **often used in generic code**
  - static_assert to check argument properties
  - SFINAE with enable_if
  - determining noexcept status
    - if constexpr (is_nothrow_movable<T>)
  - when type is not specified (auto variables) used with decltype(var)
- **classic implementation used inheritance from either true_type and false_type**
  - C++17: variable templates for _v versions

```
void demonstrate_type_queries(){
  using namespace std;
  ASSERT(is_integral_v<int>);
  ASSERT(not is_integral_v<double>);
  ASSERT(is_reference_v<int&>);
  ASSERT(not is_object_v<
    decltype(demonstrate_type_queries)>);
  ASSERT(is_object_v<int>);
  ASSERT(not is_object_v<int&>);
}
```

```
template <typename T>
struct Sack{
  static_assert(std::is_object_v<T> && !std::is_pointer_v<T>,
      "you can not use Sack with references or pointers");
};
Sack<int> sack;
//Sack<int*> ptrsack;// does not compile
//Sack<int&> refsack;// does not compile
```

© Peter Sommerlad

# Traits - compile-time-meta-programming <type_traits> - type computations

- **compute new types ..._t**
- **get to the template argument's guts**
  - remove_xxxx_t, decay_t
- **adapt integral types**
  - make_unsigned_t, make_signed_t
- **build up needed types in generic code**
  - add_xxx_t
- **classic versions (withou _t) exist, but you**
  - `using S=typename make_signed<U>::type`

```
using X=int const volatile[5];
using X1=remove_all_extents_t<X>;
ASSERT((is_same_v<X1,int const volatile>));
using X2=remove_cv_t<X1>;
ASSERT((is_same_v<X2,int>));
using RCV=int const volatile &; // cv ref to plain
ASSERT((is_same_v<int,decay_t<RCV>>));
using FR=void(&)(int); // func to funcptr
ASSERT((is_same_v<void(*)(int),decay_t<FR>>));
using AR=int const [42]; // array to ptr
ASSERT((is_same_v<int const *,decay_t<AR>>));

using I=decltype(42L);
using U=make_unsigned_t<I>;

using Tref=add_lvalue_reference_t<T>;
using Tcref=add_const_t<Tref>;
using Tptr=add_pointer_t<T>;
```

© Peter Sommerlad

# Empty class speciality: EBO Empty Baseclass Optimization

- **a class without members has at least size 1**
- **but not if it is used as a base class**
  - unless the derived type starts with a member of the same type
- **Often used to optimize away size**
  - see uniqe_ptr with default_delete or with my suggested default_free class instead of using a function pointer for free
  - also good for (CRTP-)Mix In classes, so they do not enlarge the object unnecessarily
- **C++20 adds that possibility even for "empty" members**
  - `[[no_unique_address]]` attribute

```
struct empty{};
static_assert(sizeof(empty)>0,
  "there must be something");

struct plain{
  int x;
};
static_assert(sizeof(plain)==sizeof(int),
  "no additional overhead");

struct combined : plain, private empty{
};
static_assert(sizeof(combined)==sizeof(plain),
  "empty base class should not add size");
```

© Peter Sommerlad

# When EBO does not work

- **a class without members has at least size 1**
- **but not if it is used as a base class**
  - unless the derived type starts with a member of the same type
  - each subobject of the same type must then have a unique address
- **For EBO to work nicely, have the first base hold the member(s) and further bases refer to it**
- **In addition use CRTP to ensure that each type differs**

```
struct empty{};
static_assert(sizeof(empty)>0
  && sizeof(empty)<sizeof(int),
  "there should be something");

struct ebo : empty{
  empty e;
  int i; // aligned to int
};
static_assert(sizeof(ebo)==2*sizeof(int),
  "ebo must not work");

struct noebo: empty{
  ebo e;
  int i;
};
static_assert(sizeof(noebo)==4*sizeof(int),
  "subojects must have unique addresses");
```

© Peter Sommerlad

# A glimpse of PSST (Peter's Simple Strong Typing) - EBO and CRTP-Mix-ins

```
template <typename V, typename TAG>
struct strong {
  using value_type=V;
  V val;
};

template <typename U>
struct Eq{
  friend constexpr bool
  operator==(U const &l, U const& r) noexcept {
    auto const &[vl]=l;
    auto const &[vr]=r;
    return {vl == vr};
  }
  friend constexpr bool
  operator!=(U const &l, U const& r) noexcept {
    return !(l==r);
  }
};
template <typename U>
struct Inc{
  friend constexpr auto operator++(U &rv) noexcept {
    auto &[val]=rv;
    ++val;
    return rv;
  }
  friend constexpr auto operator++(U &rv,int) noexcept {
    auto res=rv;
    ++rv;
    return res;
  }
};
template <typename U>
struct Out {
  friend std::ostream&
  operator<<(std::ostream &l, U const &r) {
    auto const &[v]=r;
    return l << v;
  }
};
template <typename U, template <typename ... > class ... BS>
struct ops:BS<U> ... {};

struct WaitC:strong<unsigned,WaitC>
          ,ops<WaitC,Eq,Inc,Out>{};
static_assert(sizeof(unsigned)==sizeof(WaitC));
void testWaitCounter(){
  WaitC c{};
  WaitC const one{1};
  ASSERT_EQUAL(WaitC{0},c);
  ASSERT_EQUAL(one,++c);
  ASSERT_EQUAL(one,c++);
  ASSERT_EQUAL(3,c.val);
}
```

aggregate

structured bindings

DANGER

delete via
base pointer

CRTP and
EBO Mixin

no overhead

# EBO in action: unique_ptr for C-allocated pointers

- **My ACCU 2017 Lightning talk**

```
inline std::string plain_demangle(char const *name){
   if (!name) return "unknown";
   char const *toBeFreed = abi::__cxa_demangle(name,0,0,0);
   std::string result(toBeFreed?toBeFreed:name);
   ::free(const_cast<char*>(toBeFreed));
   return result;
}
```

```
inline std::string plain_demangle(char const *name){
   if (!name) return "unknown";
   std::unique_ptr<char const, decltype(&std::free)>
   toBeFreed { abi::__cxa_demangle(name,0,0,0), &std::free};
   std::string result(toBeFreed?toBeFreed:name);
   return result;
}
```

```
struct free_deleter{
   template <typename T>
   void operator()(T *p) const {
      std::free(const_cast<std::remove_const_t<T>*>(p));
   }
};
template <typename T>
using unique_C_ptr=std::unique_ptr<T,free_deleter>;

static_assert(sizeof(char *)==sizeof(unique_C_ptr<char>),"");
// compiles!

inline std::string plain_demangle(char const *name){
   if (!name) return "unknown";
   unique_C_ptr<char const>
      toBeFreed {abi::__cxa_demangle(name,0,0,0)};
   std::string result(toBeFreed?toBeFreed.get():name);
   return result;
}
```

© Peter Sommerlad

# "Empty" Adapters - possible, but requires discipline!

- **"invalid" inheritance, sometimes violating Liskov Substitution Principle**
    - but OK, if only extending or adapting functionality and never sliced to base class
    - inherits constructors from base - C++11 made those adapters much more practical
- **requires discipline in use, should never implicitly "downgraded" (upcasted)**
    - slicing harmful then, beware of use in code taking the base class type as parameter
    - If you use this to strengthen the invariant, e.g., a SortedVector inheriting from std::vector, very high discipline required, better wrap then!

```cpp
template<typename T, typename CMP=std::less<T>>
class indexableSet : public std::set<T,CMP>{
  using SetType=std::set<T,CMP>;
  using size_type=int;
public:
  using std::set<T,CMP>::set;

  T const & operator[](size_type index) const {
    return at(index);
  }
  T const & at(size_type index) const {
    if (index < 0) index += SetType::size();
    if (index < 0 || index >= SetType::size())
        throw std::out_of_range{"indexableSet:"};
    return *std::next(this->begin(),index);
  }
  T const & front() const {
    return at(0);
  }
  T const & back() const {
    return at(-1);
  }
};
```

**DANGER**

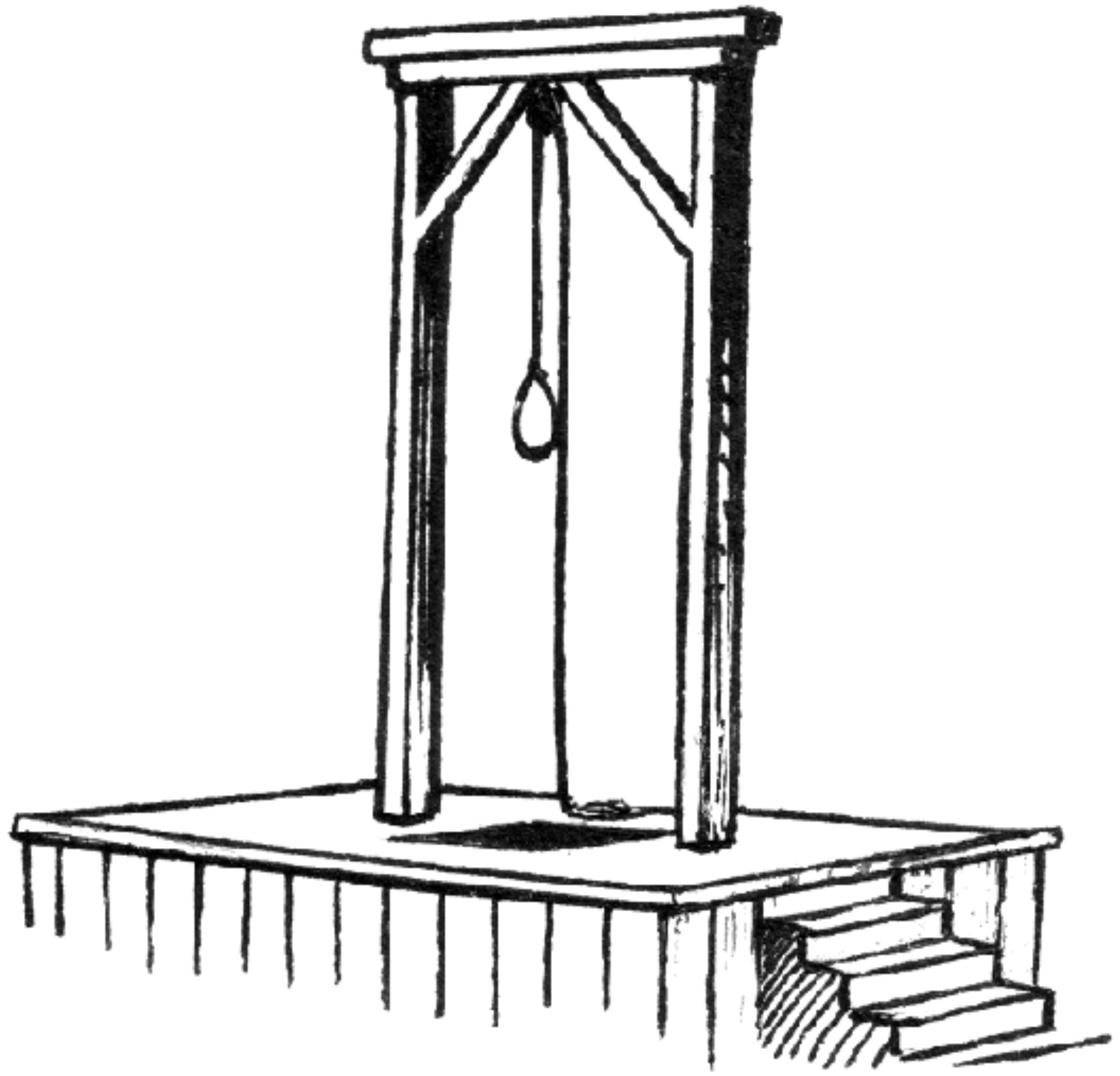
**delete via base pointer**

© Peter Sommerlad

"I just wanted to point to something"
Jonathan Müller (@foonathan), ACCU 2018

"Potentially Dangling Object Types or Potentially Dangling Types describe objects that depend on the lifetime of other referred objects. If a referred object's lifetime ends before the referring object, one risks undefined behavior."
(paraphrased from WG21-SG12/WG23 workshop in Kona 2019)

Cevelop
Your C++ deserves it

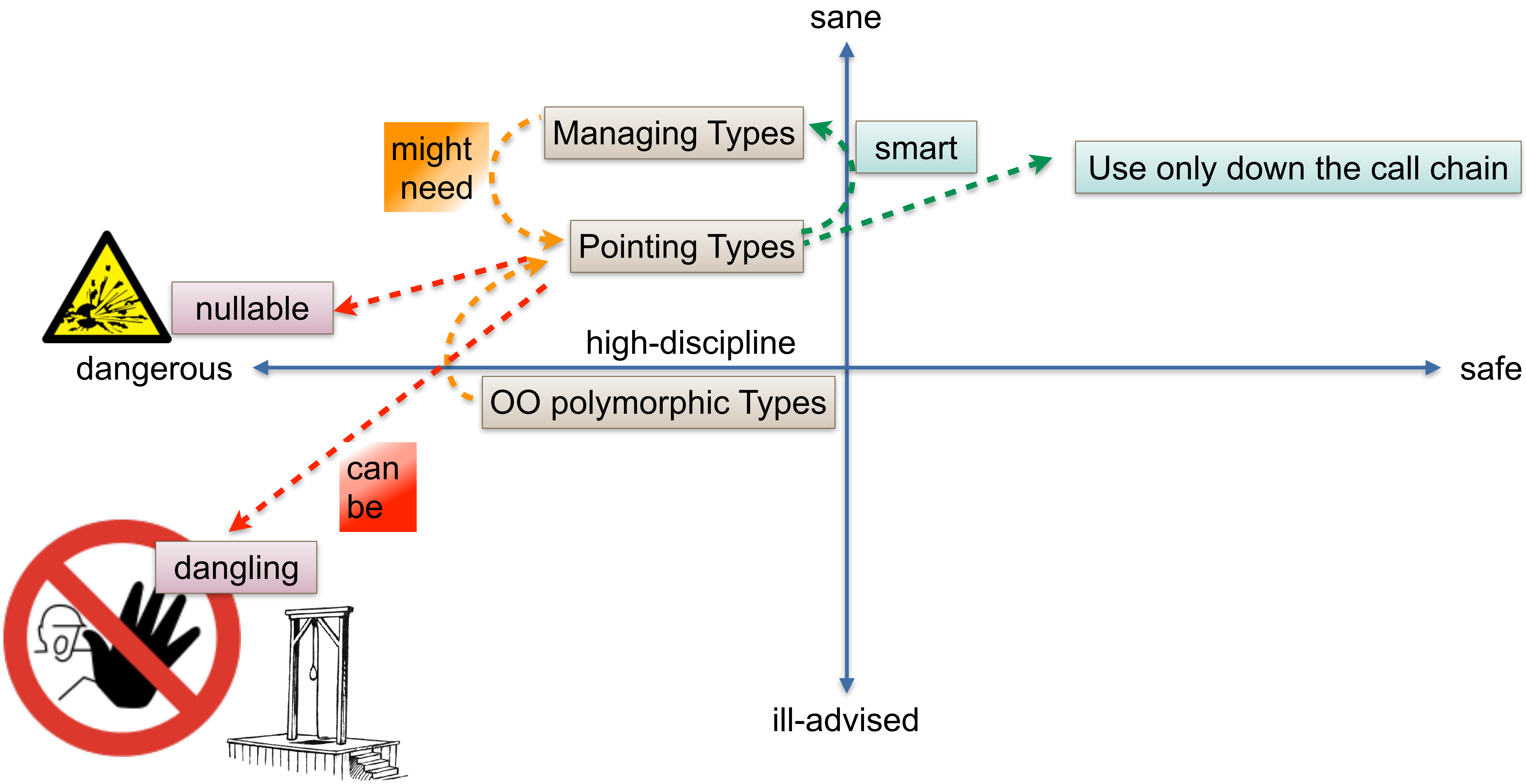


HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

© Peter Sommerlad

# Dimensions Safety and Sanity… and "pointing" types



sane

Managing Types

smart

Use only down the call chain

might need

Pointing Types

nullable

high-discipline

dangerous

safe

OO polymorphic Types

can be

dangling

ill-advised

© Peter Sommerlad

# "Pointing" Class Types

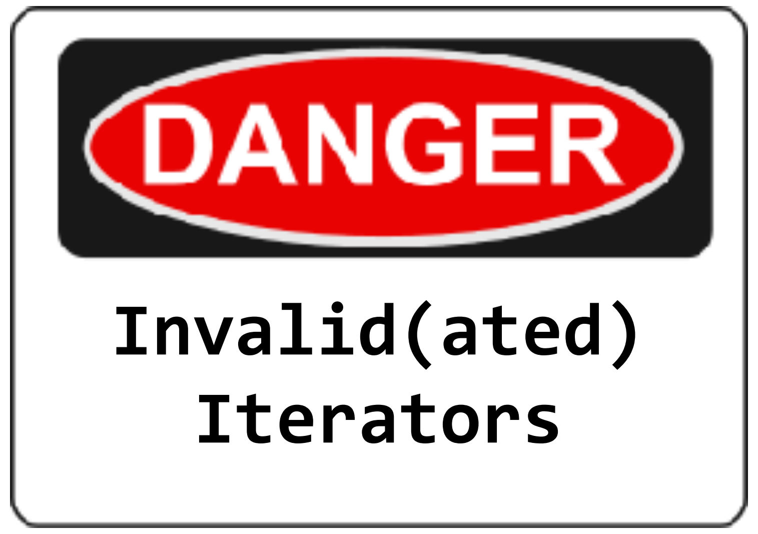
- C++ allows to define types that refer to other objects
- This means life-/using-time of the referring object needs not to extend the lifetime of the referred
- While often Regular, those types are not Value Types
  - they do not exist "out of time and space"

- Iterators
- Pointers
- Reference Wrapper
- Views and Spans (std::string_view!)

© Peter Sommerlad

# Iterators - Regular but not Values - they are **potentially dangling types**

- **Iterators satisfy concept Regular<T>, except for the need of DefaultConstructible**
  - istream(buf)_iterators have a special "eof" value, that is default constructed
- **Most iterators refer to other objects in containers**
  - relationship to the "pointed to" object as well as the container
  - changing the container can invalidate an iterator, but not always
  - dual role: reference to an object (e.g., find() result) and iteration



- **special iterator values (non-dereferencable):**
  - past the end-of-sequence iterator (end()) or before begin-of-sequence (forward_list::before_begin())
  - "singular" iterators (nullptr)
  - invalidated iterators due to changes in the container
  - Do not rely on iterator staying valid if a container's content can change

Usually invalid iterators can not be detected: UB

© Peter Sommerlad

# Special "pointer": std::reference_wrapper (non-null, non-owning, assignable, callable)

- **role: re-assignable lvalue (const) reference**
  - is not "nullable"! But can be dangling!
- **can be used for class members to keep class "regular"**
  - T& as a member disables assignment
- **can be used in container to refer to elements in other container**
  - use a container of (indices) into other container
- **automatically converts to reference**
  - or access via get()
- **wraps function references**
  - overloads operator()
- **Factory functions: std::ref(T&), std::cref(T const&)**

```
template <class T>
class reference_wrapper {
public:
  // types
  typedef T type;

  // construct/copy/destroy
  reference_wrapper(T& ref) noexcept : _ptr(std::addressof(ref)) {}
  reference_wrapper(T&&) = delete;
  reference_wrapper(const reference_wrapper&) noexcept = default;

  // assignment
  reference_wrapper& operator=(const reference_wrapper& x)
    noexcept = default;

  // access
  operator T& () const noexcept { return *_ptr; }
  T& get() const noexcept { return *_ptr; }

  template< class... ArgTypes >
  std::invoke_result_t<T&, ArgTypes...>
    operator() ( ArgTypes&&... args ) const {
    return std::invoke(get(), std::forward<ArgTypes>(args)...);
  }

private:
  T* _ptr;
};
```

© Peter Sommerlad

# Smart Pointers - are not necessarily "potentially dangling types"

- **observer_ptr<T> better: jss::object_ptr<T>**
  - "borrows" object, does not own pointee
  - library fundamentals TS v2 (not std)
  - [object_ptr - a safer replacement for raw pointers]
- **unique_ptr<T> - can not dangle!**
  - owns pointee, cleans afterwards
- **shared_ptr<T>, weak_ptr<T> - can not dangle!**
  - shared ownership
  - overhead for atomic counting
  - may "pseudo-leak", even when object is deleted

```
template <typename T>
using observer_ptr=T *;
```

My current recommendation:

- prefer unique_ptr<T> for heap-allocated objects
- for sharing keep unique_ptrs in a managing container and use references or reference_wrapper (some would say to use T* pointers)
- absolutely NO plain pointers with arithmetic (as in C)

© Peter Sommerlad

# Views and Spans - Range-References for contiguous memory

- **References to contiguous sequences (e.g., from std::vector, std::array, std::string)**
- **Naming is contentious**
  - does a view allow changing the elements? --> span does
- **today: std::string_view**
  - std::string, std::array<char, N>, std::vector<char>
  - caveat: almost all of std::string bloated interface, except for mutation of characters
  - pure read-only, idea to replace (char const *) function parameters, but existing overloads :-(
- **C++20 (and core guidelines support library): span<T, int Extent>**
  - contention: static (compile-time) vs. dynamic extent (run-time)
  - allows mutation of elements
  - replacement for (T* , size_t len) function interfaces (C)

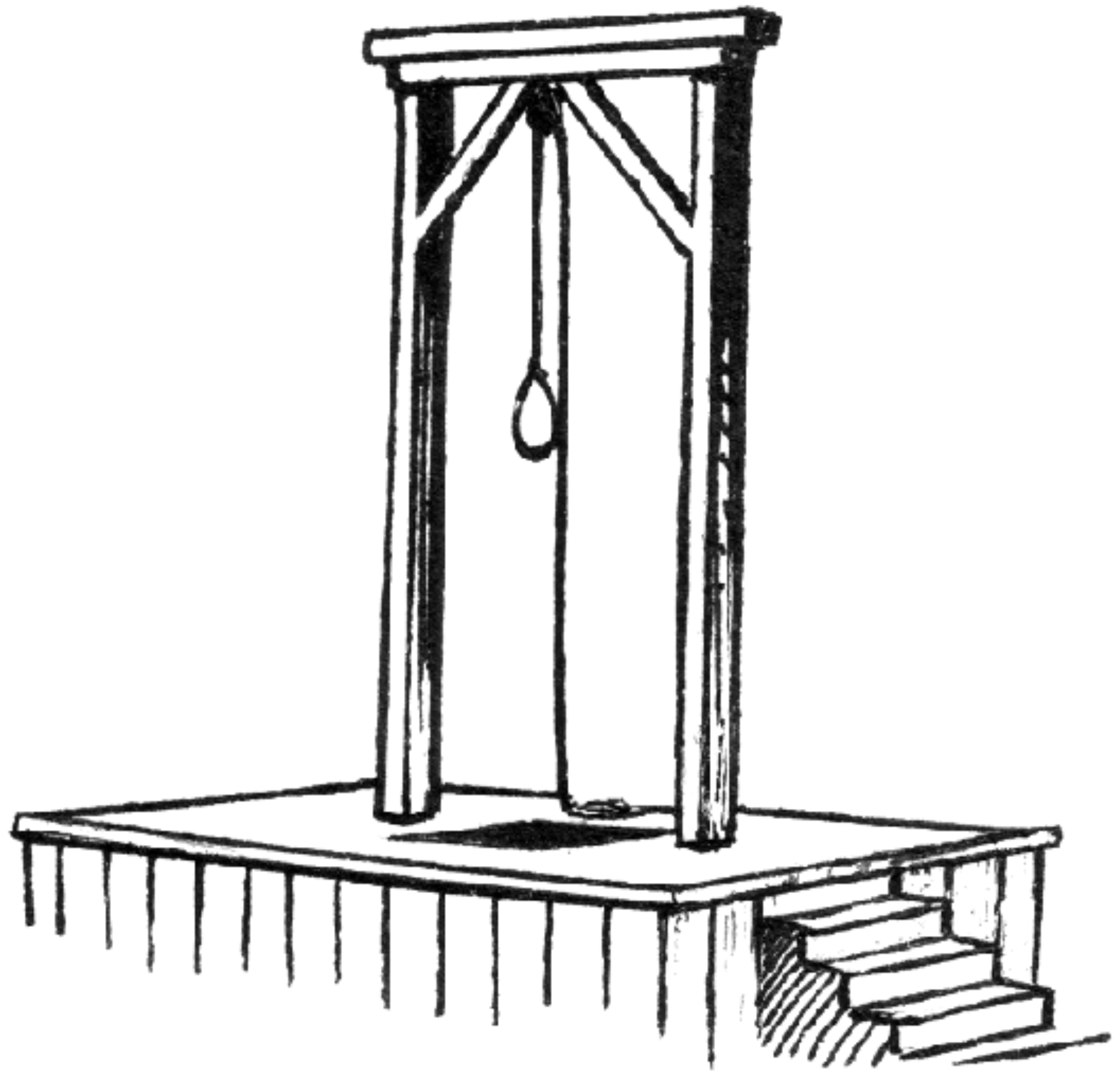
Dangling
References

High-performance computing people define span<> to support multi-dimensional array views with mutable elements (P0546)

© Peter Sommerlad

# Where should I use string_view? - also a potentially dangling type

- **As a parameter type for functions that do not copy, save or change a string**
  - If read-only string processing is required
- **enables calling with C-style (char array) strings and std::string**
  - safer than (char const *)
  - better performance than (std::string const &)
  - beware of generic overloads when replacing existing APIs
    - might need overloads for all available character types (string_view, wstring_view) - no CharT deduction possible
    - I tried for the standard and failed!
- **In practice much less useful than I originally thought**
  - std::string pass-by-value often better when serious processing is required
- **Do never return std::string_view!**



DANGER

Dangling string_view

© Peter Sommerlad

# C-style pointers: T*, T const *, T[] - also a potentially dangling types

- **Always define pointer variables const**
  - absolutely no pointer arithmetic!!!!!
  - especially for pointer parameters
- **Sidestep plain C-style pointers completely in user code**
- **Absolutely NO C-style arrays, because they are pointers in disguise**
  - they degenerate to pointers and require pointer arithmetic!
  - even built-in operator[] is pointer arithmetic!

```
int demo(int *const pi){
  //*pi++;
  (*pi)++;
  return *pi;
}
```

```
void dont_demo(int *const pi){
   1[pi]=42;
   pi[0]=41;
}
void testDont(){
   std::array<int,2> a{};
   dont_demo(a.data());
   std::initializer_list<int> exp{41,42};
   ASSERT_EQUAL_RANGES(begin(exp),end(exp),begin(a),end(a));
}
```

© Peter Sommerlad

# What to do about it?

- **All "pointing" Types live in the "dangerous" half**
- **High programmer discipline required**
- **Unfortunately code compiles**
  - often for backward compatibility
  - rules for iterator invalidation are subtle and rely on knowing implementation details
    - changing a container breaks code
    - Do not rely on iterator staying valid if a container's content can change
- **Ideas exist for static analysis (-> Herb Sutter)**
  - it is safe to pass them down the call chain
  - it is often unsafe to use them if you do not control the lifetime of the pointee

© Peter Sommerlad

# Referring stuff obtained from temporaries is dangerous

- **problem with reverse adapter for range for (CPPCon 2018 ⚡ Talk)**
  - https://github.com/PeterSommerlad/ReverseAdapter
  - init-statements with additional variable is just too ugly, IMHO
- **Just an idea (may be worth a ISO C++ paper?)**
  - provide deleted overloads for begin(), end() etc for rvalue references.
  - might break already wrong code
  - members returning elements by reference should return by value for temporaries

Dangling References

Invalidated Iterators

```
void testTemporaryArrayAccess(){
   ASSERT_EQUAL(2,(std::array{1,2,3}).at(1));
   int &i = std::array{2,3}[0];
   i=1; // UB
}
void testBeginTemporaryShouldNotCompile(){
   auto it = std::array{1,2,3}.begin();
   ASSERT_EQUAL(1,*it);
}
```

```
constexpr iterator
begin() & noexcept
{ return iterator(data()); }
constexpr const_iterator
begin() const & noexcept
{ return const_iterator(data()); }
constexpr iterator
begin() && noexcept  = delete;
```

```
constexpr reference
operator[](size_type __n)  & noexcept
{ return _AT_Type::_S_ref(_M_elems, __n); }
constexpr const_reference
operator[](size_type __n) const & noexcept
{ return _AT_Type::_S_ref(_M_elems, __n); }
constexpr value_type
operator[](size_type __n) && noexcept
{ return std::move(_M_elems[__n]); }
```

© Peter Sommerlad

Managing stuff

"monomorphic object types"
  -- Richard Corden, PRQA

"SBRM - scope-based resource management"
  -- a better name for RAII

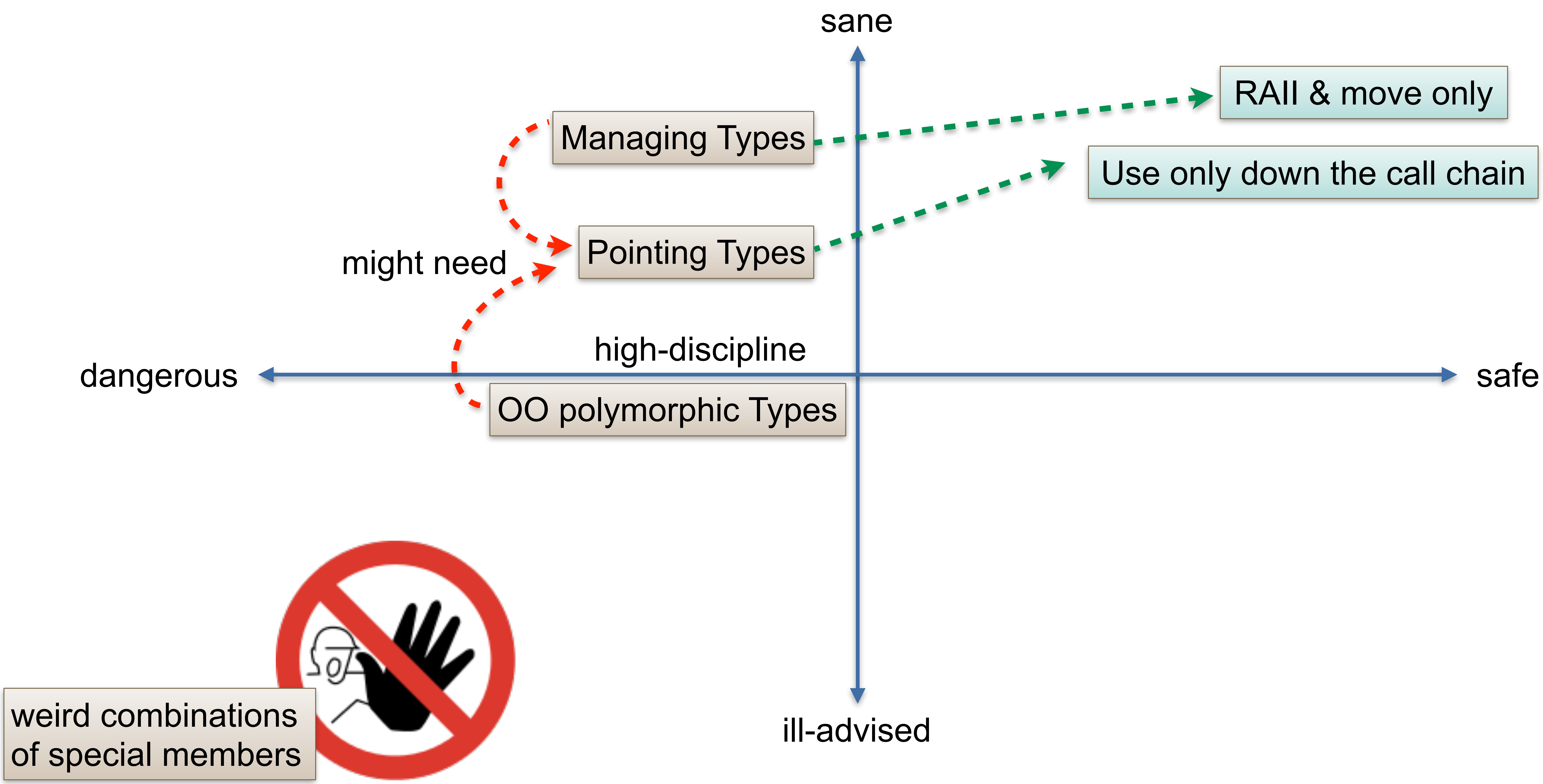
Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

© Peter Sommerlad

# Dimensions Safety and Sanity…



sane

RAII & move only

Managing Types

Use only down the call chain

Pointing Types

might need

high-discipline

dangerous

safe

OO polymorphic Types

weird combinations of special members

ill-advised

© Peter Sommerlad

# Managing types complexity staging

- **Common to Managing types**
  - define "interesting" destructor: `~manager() { /* clean up stuff */}`
- **0: locally usable SBRM (e.g., `std::lock_guard`)**
  - Rule of DesDeMovA: `manager& operator=(manager &&) noexcept=delete;` 🙋‍♀️
  - No movability implies also no copyability
  - C++17: can still return from factory if needed
- **1: unique - move-only type (e.g., `std::unique_ptr`)**
  - requires a **sane moved-from state** for transfer of ownership, copy operations implicitly deleted
- **N: value type (e.g., `std::vector`)**
  - requires duplicatable resource (aka memory)

Expert-level
Experience!

© Peter Sommerlad

# Manager Design Pattern: Monomorphic Object Types (SBRM)

- **Instances of monomorphic object types have significant identity (they are not values)**
- **Copying and assignment is prohibited**
  - Factories can still return by value from a temporary (C++17!)
  - Apply "Rule of DesDeMovA"
- **Passed by Reference (or Pointer-like type)**
  - "long" lifetime, allocated high-up the call hierarchy or on heap
- **No virtual members, no inheritance (except for mix-ins)**
- **Roles**
  - manage other objects, i.e., contain a container of something: vector<unique_ptr<T>> as member
  - wrap hardware or stateful I/O
  - encapsulate other stateful behavior, e.g., context of State design pattern, Builder, Context Object

```
struct ScreenItems{
  void add(widget w){
    content.push_back(std::move(w));
  }
  void draw_all(screen &out){
    for(auto &drawable:content){
      drawable->draw(out);
    }
  }
private:
  ScreenItems& operator=(ScreenItems &&) noexcept
     =delete; // all others deleted, except default
  widgets content{};
};
static_assert(!std::is_copy_constructible_v<ScreenItems>,"no copying");
static_assert(!std::is_move_constructible_v<ScreenItems>,"no moving");
ScreenItems makeScreenItems(){
  return ScreenItems {}; // must be a temporary
}
```

© Peter Sommerlad

# Use existing RAII (Resource Acquisition Is Initialization) for SBRM

- **OK, make_unique() (**and make_shared) **for heap allocation.**
- **What else?**
- **Use std-library RAII classes, e.g., string, vector, fstream, ostringstream, thread, unique_lock**
- **Use boost-library RAII classes, if needed, e.g., boost.asio’s tcp::iostream**

- **Don’t write your own generic RAII!**
- **~~wait for unique_resource<T,D>: http://wg21.link/p0052~~**
  - You can help with me [https://github.com/PeterSommerlad/scope17](https://github.com/PeterSommerlad/scope17)

© Peter Sommerlad

# Dynamic Polymorphism

"inheritance is the base class of Evil"
 -- Sean Parent, Adobe

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

© Peter Sommerlad

# Do you 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 a bug in the standard.

© Peter Sommerlad

# Making a OO base class T non-copyable: T& operator=(T&&) noexcept=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.

© Peter Sommerlad

# Rule of DesDeMovA: T& operator=(T&&) noexcept=delete;

## DesDeMovA 🙋‍♀️

Rule of if<br>
**Des**tructor defined<br>
**De**leted<br>
**Mov**e **A**ssigment

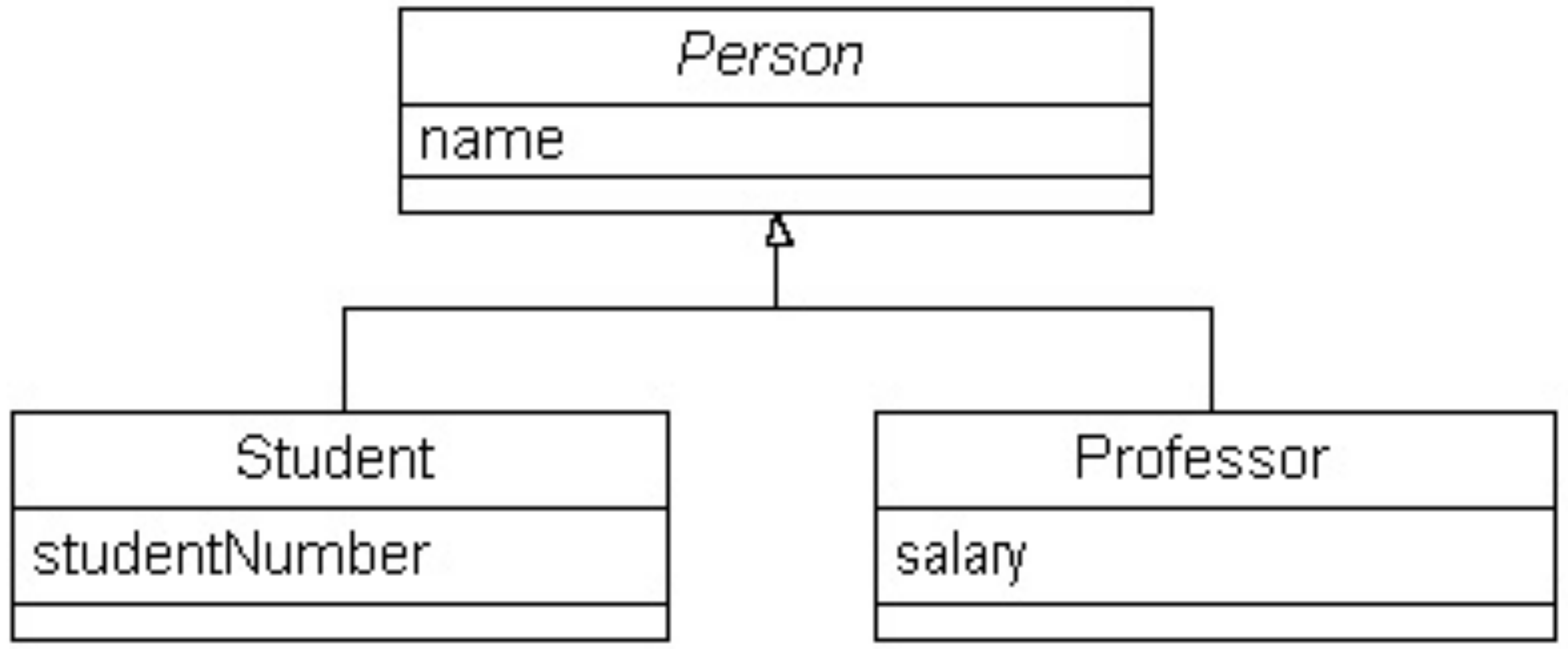
What you write

| | default constructor | | py nment | move constructor | move assignment |
|---|---|---|---|---|---|
| nothing | defaulted | | ulted | defaulted | defaulted |
| any constructor | not declared | | ulted | defaulted | defaulted |
| default constructor | user declared | 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<br>
Note: Getting the defaulted special members denoted with a (!) is a bug in the standard.

© Peter Sommerlad

# Polymorphic Object Types -- think thrice about using virtual!



good OO design?

- **Base in class in hierarchy defines abstraction**
  - usually abstract (pure virtual destructor)
- **Instances of polymorphic object types have important identity**
- **Copying and assignment is prohibited (implicitly or explicitly) - non-Regular types**
- **Passed by Reference (or Pointer-like type)**
  - "long" lifetime, allocated up in the call hierarchy (best) or on the heap (doable)
- **Virtual member functions and (pure) virtual destructor in base class**
  - subclasses should not add additional virtual members, define pure virtual destructor of base
- Most other attempts with multiple layers of inheritance or even multiple inheritance are often futile

© Peter Sommerlad

# Polymorphic Hierarchy example: Composite Design Pattern

```
struct drawable {
  virtual ~drawable()=0;
  virtual void draw(screen& on)=0;
protected:
  drawable&
operator=(drawable&&)noexcept=delete;
  // prohibit move and copy
};
drawable::~drawable()=default;
```

**DesDeMovA** 🙋

```
struct composite:drawable{
  composite()=default;
  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{};
};
```

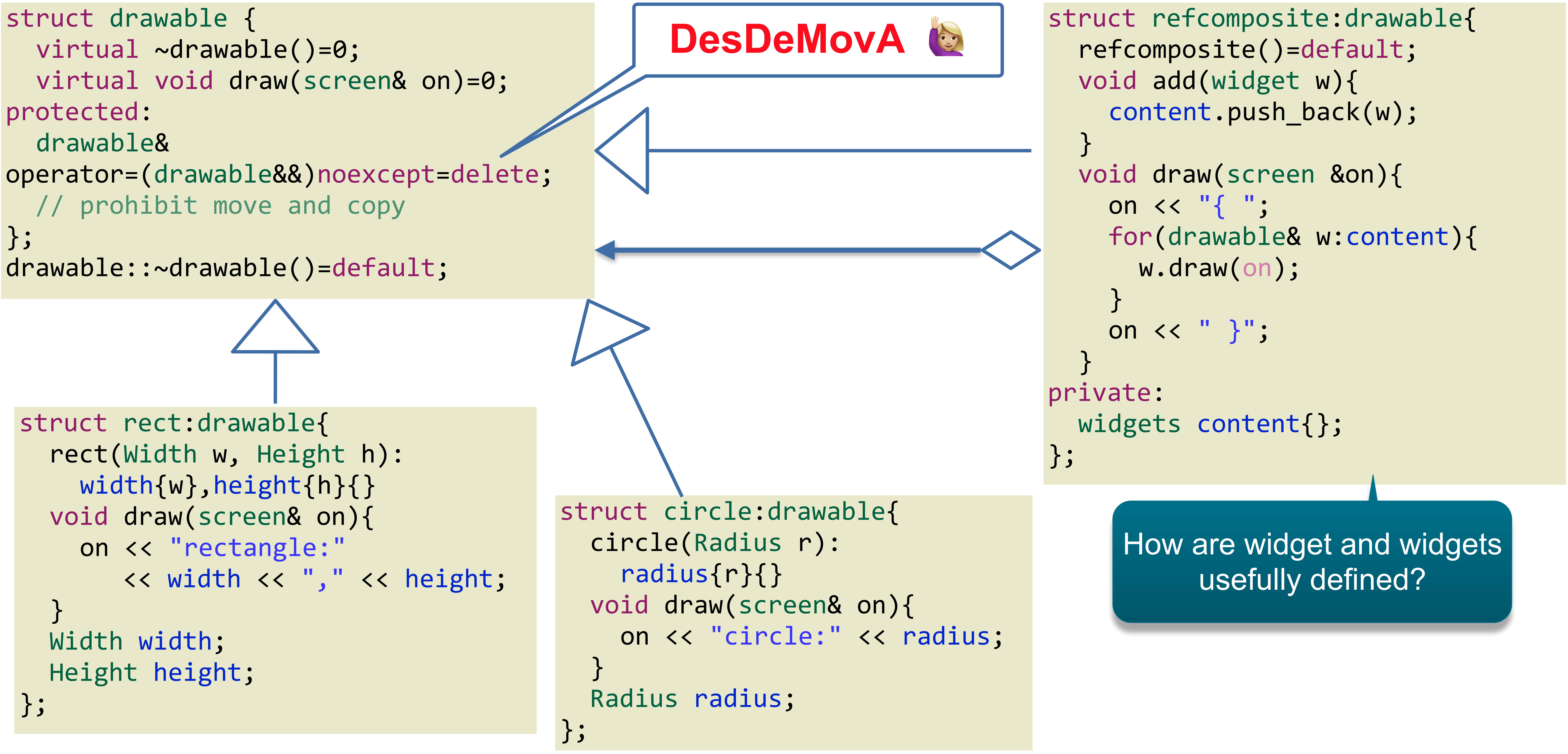
```
struct rect:drawable{
  rect(Width w, Height h):
    width{w},height{h}{}
  void draw(screen& on){
    on << "rectangle:"
       << width << "," << height;
  }
  Width width;
  Height height;
};
```

```
struct circle:drawable{
  circle(Radius r):
    radius{r}{}
  void draw(screen& on){
    on << "circle:" << radius;
  }
  Radius radius;
};
```

How are widget and widgets usefully defined?

© Peter Sommerlad

# Polymorphic Hierarchy: Composites with references



```
struct drawable {
  virtual ~drawable()=0;
  virtual void draw(screen& on)=0;
protected:
  drawable&
operator=(drawable&&)noexcept=delete;
  // prohibit move and copy
};
drawable::~drawable()=default;
```

DesDeMovA 🙋‍♀️

```
struct refcomposite:drawable{
  refcomposite()=default;
  void add(widget w){
    content.push_back(w);
  }
  void draw(screen &on){
    on << "{ ";
    for(drawable& w:content){
      w.draw(on);
    }
    on << " }";
  }
private:
  widgets content{};
};
```

```
struct rect:drawable{
  rect(Width w, Height h):
    width{w},height{h}{}
  void draw(screen& on){
    on << "rectangle:"
       << width << "," << height;
  }
  Width width;
  Height height;
};
```

```
struct circle:drawable{
  circle(Radius r):
    radius{r}{}
  void draw(screen& on){
    on << "circle:" << radius;
  }
  Radius radius;
};
```

How are widget and widgets usefully defined?

© Peter Sommerlad

# Type-erasure Object/Value dualities

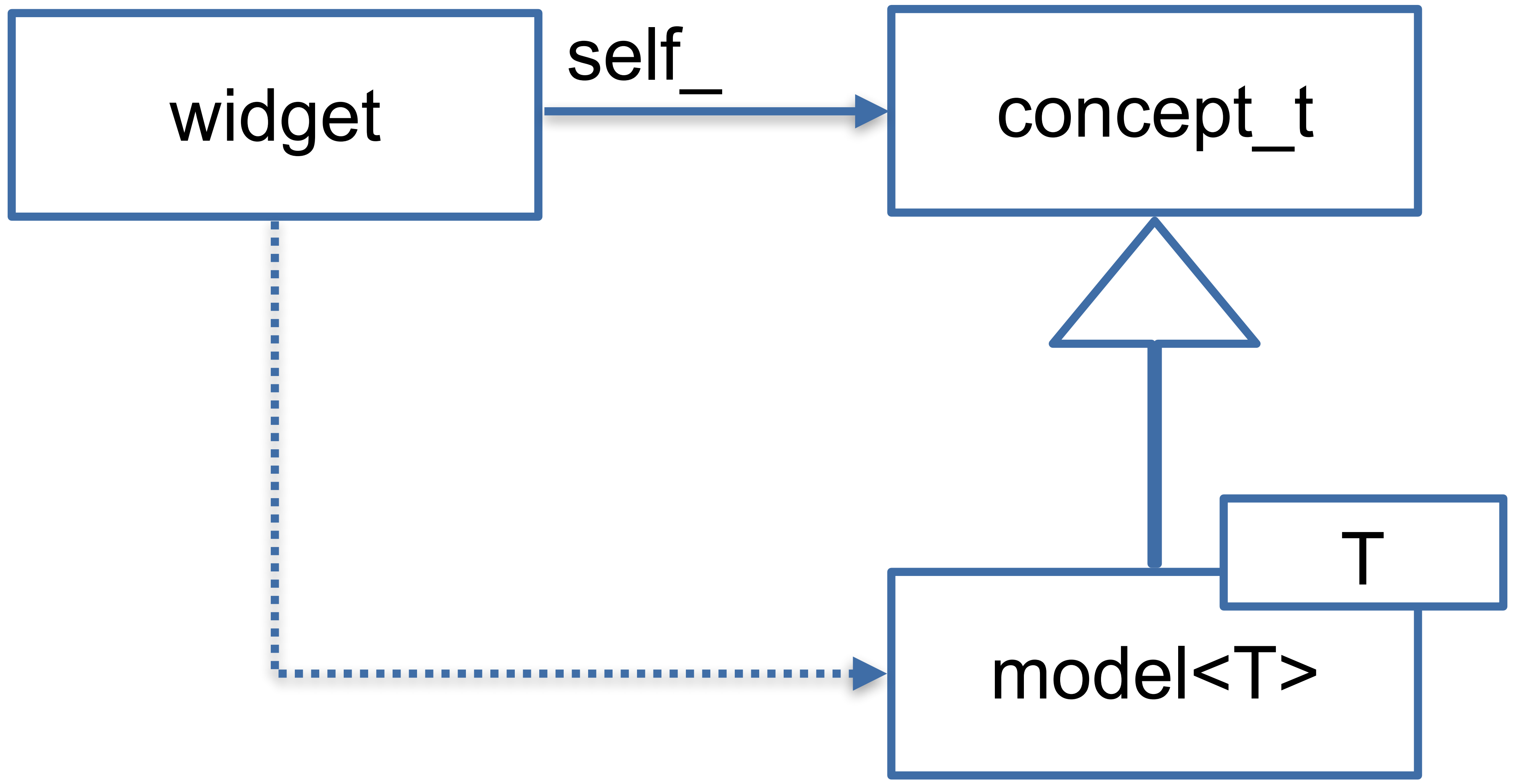
- **An observation:**
  - `std::function<ret(params)> var;` // can store any kind of function matching signature
  - How?
- **`std::any some;` can store any value type**
  - can only access what was stored
    - and can be empty
  - often better `std::variant` when when set of possible types is known
    - a variant can not be empty
      - **except under exceptional condition**

```
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));
}
```

© Peter Sommerlad

# Type-erasure-based Object/Value dualities

- **Sean Parent: dynamic Polymorphism without inheritance**
  - make polymorphic stuff regular and extendible without inheritance
  - requires also a mechanism called "type erasure"
  - combines inheritance and templates internally
  - can store arbitrary values, like any
  - and provide interface to them
- **How?**



```
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("{ circle:42,rectangle:"
   "4,2,circle:4,an_int:42,"
   "a c string, }",out.str());
}
```

© Peter Sommerlad

# Sean Parent's magical polymorphic Regular objects

```
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);
  }

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;
  };
  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);
    }
    void draw_(screen& out) const {
      draw(data_, out);
    }

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

© Peter Sommerlad

# Sean Parent's magical polymorphic Regular objects - usage

```
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;
}
```

```
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{};
};

void testRect(){
  std::ostringstream out{};
  widget r{rect{Width{2},Height{4}}};
  draw(r,out);
  ASSERT_EQUAL("rectangle:2,4",out.str());
}
```

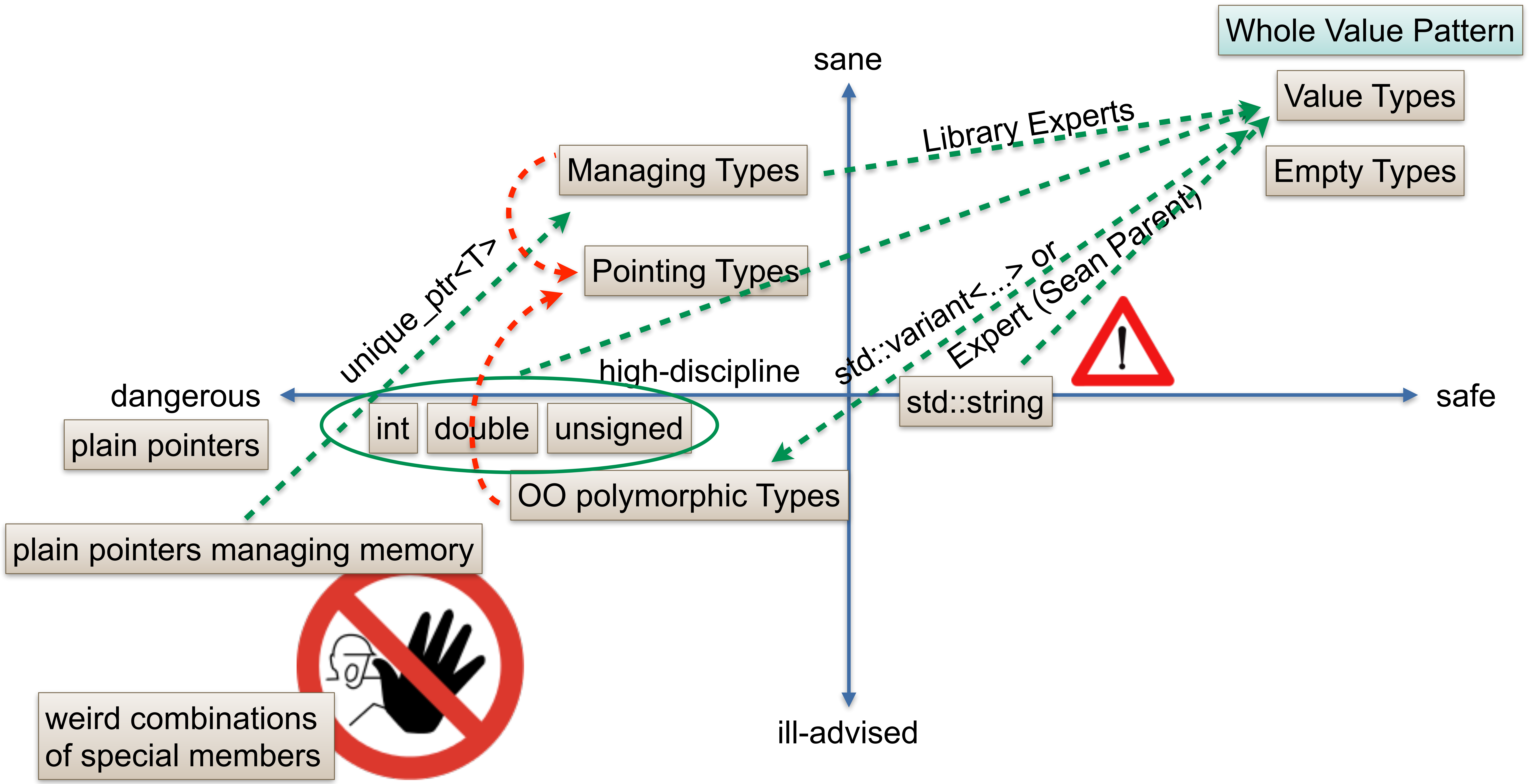
© Peter Sommerlad

# Sane and less sane combinations

| | | Some constructor | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---|---|---|---|---|---|---|---|---|
| | Aggregates | none | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| | Simple Values | yes | none /<br>=default | defaulted | defaulted | defaulted | defaulted | defaulted |
| Manager | Simple | typical | none /<br>=default | implemented | deleted | deleted | deleted | =delete 🙋‍♀️ |
| | Unique | typical | defined /<br>=default | implemented | deleted | deleted | implemented | implemented |
| | Value | yes | defined /<br>=default | implemented | implemented | implemented | implemented | implemented |
| | OO - Base | may be | may be | =default<br>virtual! | deleted | deleted | deleted | =delete 🙋‍♀️ |
| | OO & Value Sean Parent | yes | no | Expert Level - =default | Expert Level Implementation | Expert Level Implementation | Expert Level Implementation | Expert Level Implementation |

© Peter Sommerlad

# Dimensions Safety and Sanity…



© Peter Sommerlad

# Take aways 🍔 🍟

- **Learn to appreciate the C++ Type System - every cast is an indication to think & refactor!**
- **Model with Value Types almost always**
    - but be aware of the **relative vs. absolute** dimension problem in your units!
- **Wrap primitives using Whole Value, even a named simple struct communicates better than** int
- **Be aware of the required expertise and discipline for Manager types and OO hierarchies**
    - Remember **"Rule of DesDeMovA"**
- **Be very disciplined about using Pointing types, this includes references and string_view**
- **Run away from types with weird special member function combinations, even if defaulted**
    - usually they attempt to do too much or the wrong thing

© Peter Sommerlad