Advanced Developers Conference C++
Development for Professionals!

Sane and Safe C++ Class Types

Prof. Peter Sommerlad

ppedv

Sane and Safe C++ Class Types

Prof. Peter Sommerlad
Director of IFS
ADC-C++ Regensburg, May 2019

@PeterSommerlad
peter.cpp@sommerlad.ch

Typical C++ problems ?

Patricia Aas
@pati_gallardo

Following

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.
4 17

Jonathan Caves @joncaves · Mar 16
Try living in a 30+ year old code base :(
2 7

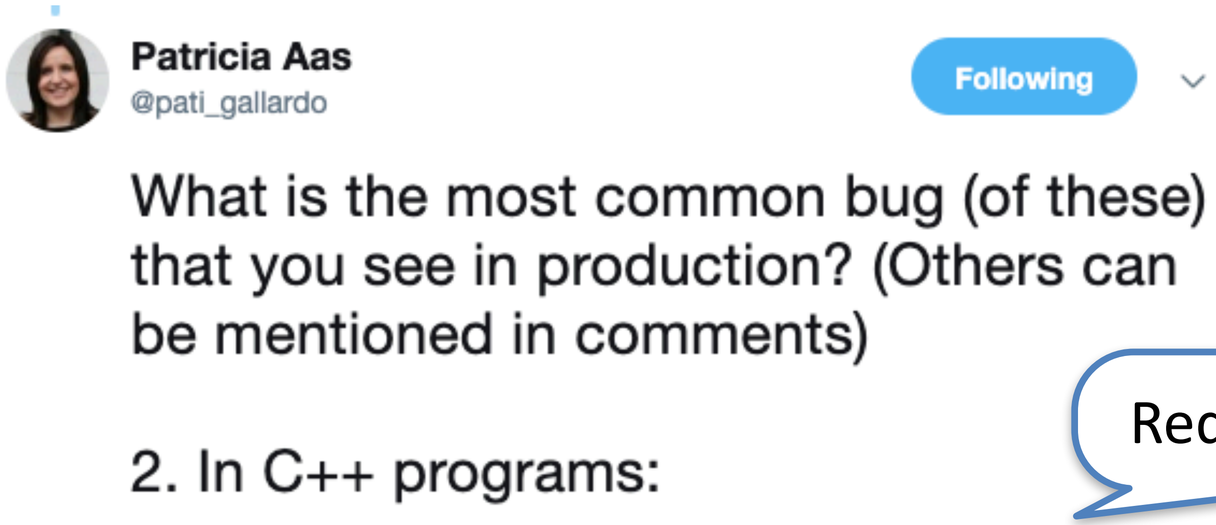
Michael Caisse @MichaelCaisse · Mar 16
I gave that up years ago. It brought no joy.
5

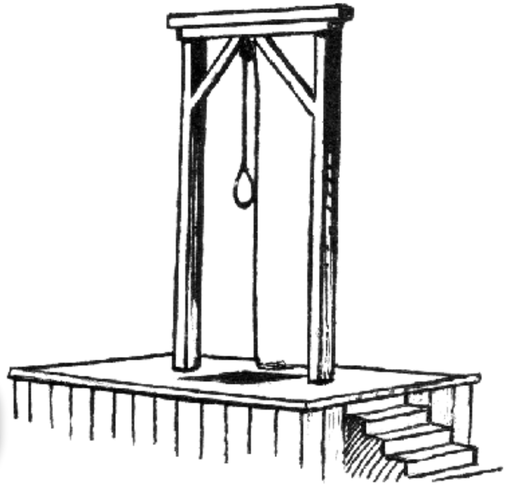
Peter Sommerlad @PeterSommerlad · Mar 16
Replying to @pati_gallardo
None of the above.
1 7

Patricia Aas @pati_gallardo · Mar 16
You have none of the bugs above?
3

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.
1 3

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





Requires Discipline!

partially avoidable: values instead of pointers, references, views

100% avoidable: unique_ptr, containers, or values

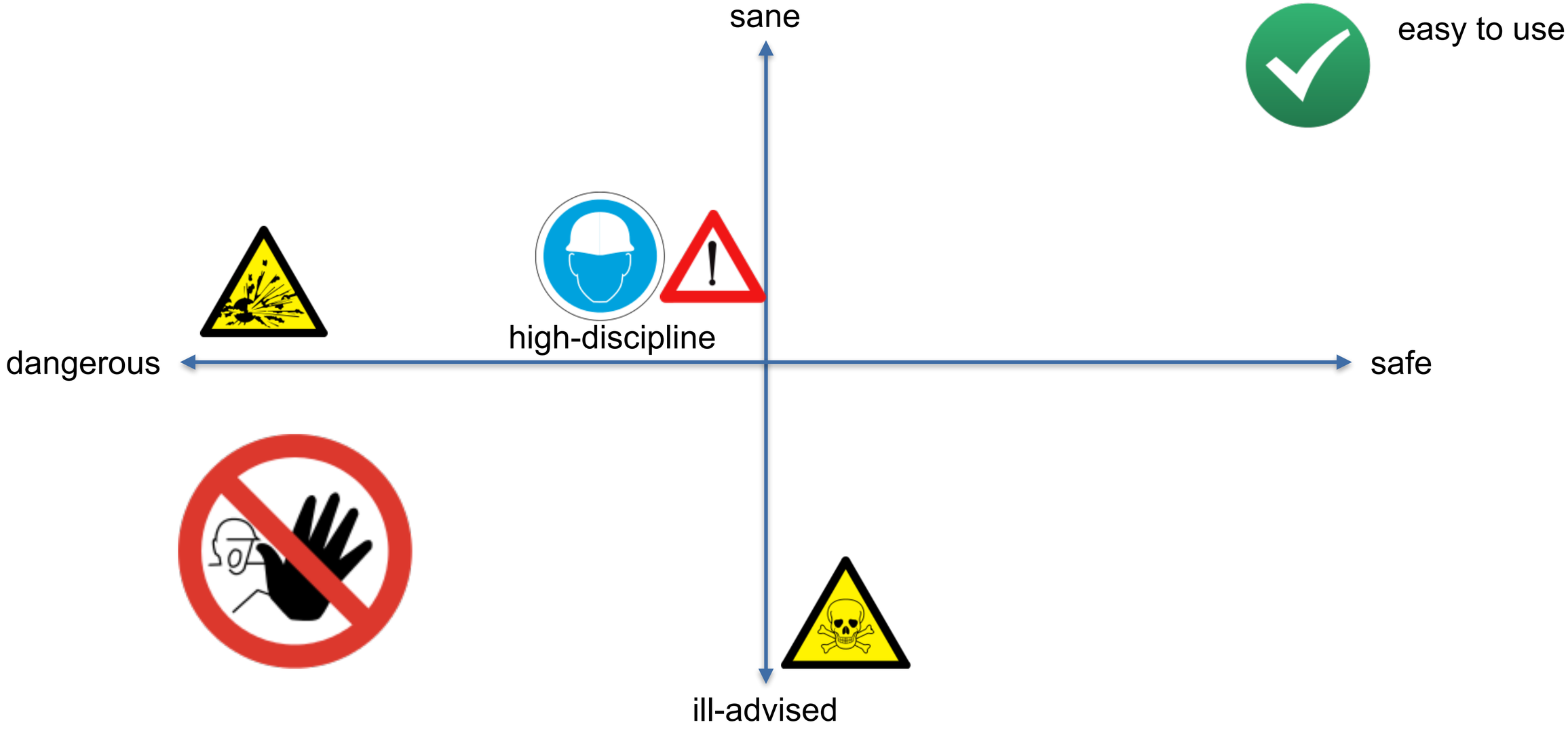
100% avoidable: unique_ptr, containers, or values

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

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

except tightly encapsulated

Dimensions Safety and Sanity...



- **Value Types**
- **Empty Types**
- **Managing Types (different flavors)**
  - 0. SBRM/RAII - scope-based resource management
  - 1. Unique resource manager (move-only)
  - N. Value-type resource manager (copyable)
- **OO-polymorphic-hierarchy Types**

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

Rules for Special Member Functions

- **Rule of Zero**
  - Write your classes that you can rely on the compiler provided ones
- **Rule of Three**
  - Scott Meyers' classic: When you define either the destructor or a copy operation, define all three
  - variation: declare copy operations private/protected to prevent slicing
- **Rule of Five/Six**
  - Scott Meyers extended for C++11: one for all, all for one (including default ctor)
- **Rule of DesDeMovA**
  - stay tuned.

Values

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

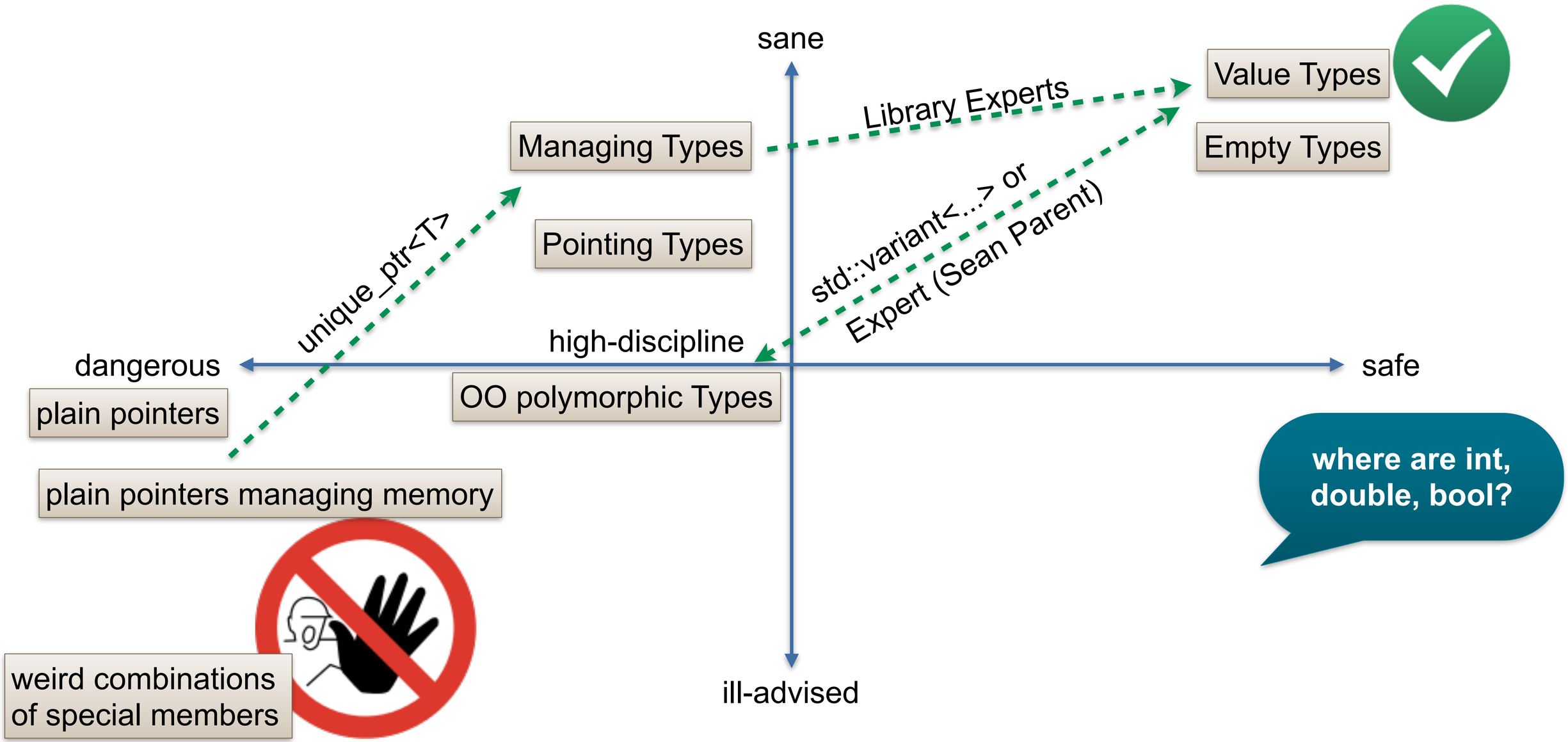
"But may be not always..."
-- Peter Sommerlad

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Value (class) Types are Sane and Safe!



Some of the Problems with **primitive** built-in types

```
ASSERT_EQUAL(0.0,sin(false));
ASSERT_EQUAL_DELTA(1.0,sin(true),0.2);
ASSERT_EQUAL(true,bool(sin(true)));
```

- **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!

C++ built-in numeric types are not good for abstracting domain values

- **multiple parameters of same type**
  - can not easily distinguish
  - people call for "named parameters"
    - IMHO: wrong approach for C++
- **type aliases do not help**
  - different for humans, not for compiler
- **Every domain or application has specific computations and thus specific types used**
  - often mapped to integers or floating point numbers without thinking
  - education problem (abstraction)
  - considered a performance issue

```
double consumption(double l, double km) {
  return l/(km/100.0);
}

void testConsumption1over1(){
  double const l { 1 } ;
  double const km { 1 } ;
  ASSERT_EQUAL(100.0, consumption(l, km));
}

void testConsumption40over500(){
  double const l { 40 };
  double const km { 500 };
  ASSERT_EQUAL(8.0, consumption(l, km));
}

void testConsumption40over500Wrong(){
  double const l { 40 };
  double const km { 500 };
  ASSERT_EQUAL(8.0, consumption(km, l)); // no check.
}
void testConsumptionEvenMoreStrange(){
  ASSERT_EQUAL(8.0,consumption(consumption(40,500),100));
}
```

```
consumption(km, l) or consumption(l,km)
```

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

it will be called wrongly!

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.*
  - *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

Do not use primitive types (int) or generic representation types (string) for your domain values!

```
double consumption(double l, double km) {
  return l/(km/100.0);
}
```

Apply "simplest strong" type wrapper: struct

- **simple aggregate types**
  - one for each domain value
  - no encapsulation
  - as efficient as normal
- **even works in C**
  - with slightly more elaborate spelling
  - structs can be passed by value
  - as efficient as primitive types
- **Disclaimer:**
  - ABI might not allow same efficiency

```
struct literGas{
  double l;
};
struct kmDriven{
  double km;
};
struct literPer100km {
  double consumption;
};

literPer100km consumption(literGas l, kmDriven km) {
  return {l.l/(km.km/100.0)}; // needs curlies
}
void testConsumption40over500(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(8.0,consumption(l,km).consumption);
}

void testConsumption40over500Wrong(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(8.0,consumption(km,l).consumption);
}
```

error: no matching function for call to 'consumption'

Simple Type Wrappers produce identical code

C++ source #1

Save/Load + Add new... CppInsights C++

```
1  struct literGas{
2      double l;
3  };
4  struct kmDriven{
5      double km;
6  };
7  struct literPer100km {
8      double consumption;
9  };
10
11 literPer100km consumption(literGas l, kmDriven km) {
12     return {l.l/(km.km/100.0)}; // needs curlies
13 }
14
```

C++ source #2

Save/Load + Add new... CppInsights C++

```
1
2
3
4
5
6
7
8
9
10
11 double consumption(double l, double km) {
12     return l/(km/100.0);
13 }
14
```

x86-64 clang 8.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 8.0.0 -O3 -Wextra -Wall -Werror

11010 .LX0: lib.f: .text // \s+ Intel Demangle

```
1  .LCPI0_0:
2          .quad   4636737291354636288     # double 100
3  consumption(literGas, kmDriven):        # @consumption(literGas, kmD
4          divsd   xmm1, qword ptr [rip + .LCPI0_0]
5          divsd   xmm0, xmm1
6          ret
```

Output (0/0) x86-64 clang 8.0.0 - 1191ms (11692B)

x86-64 clang 8.0.0 (Editor #2, Compiler #2) C++

x86-64 clang 8.0.0 -O3 -Wextra -Wall -Werror

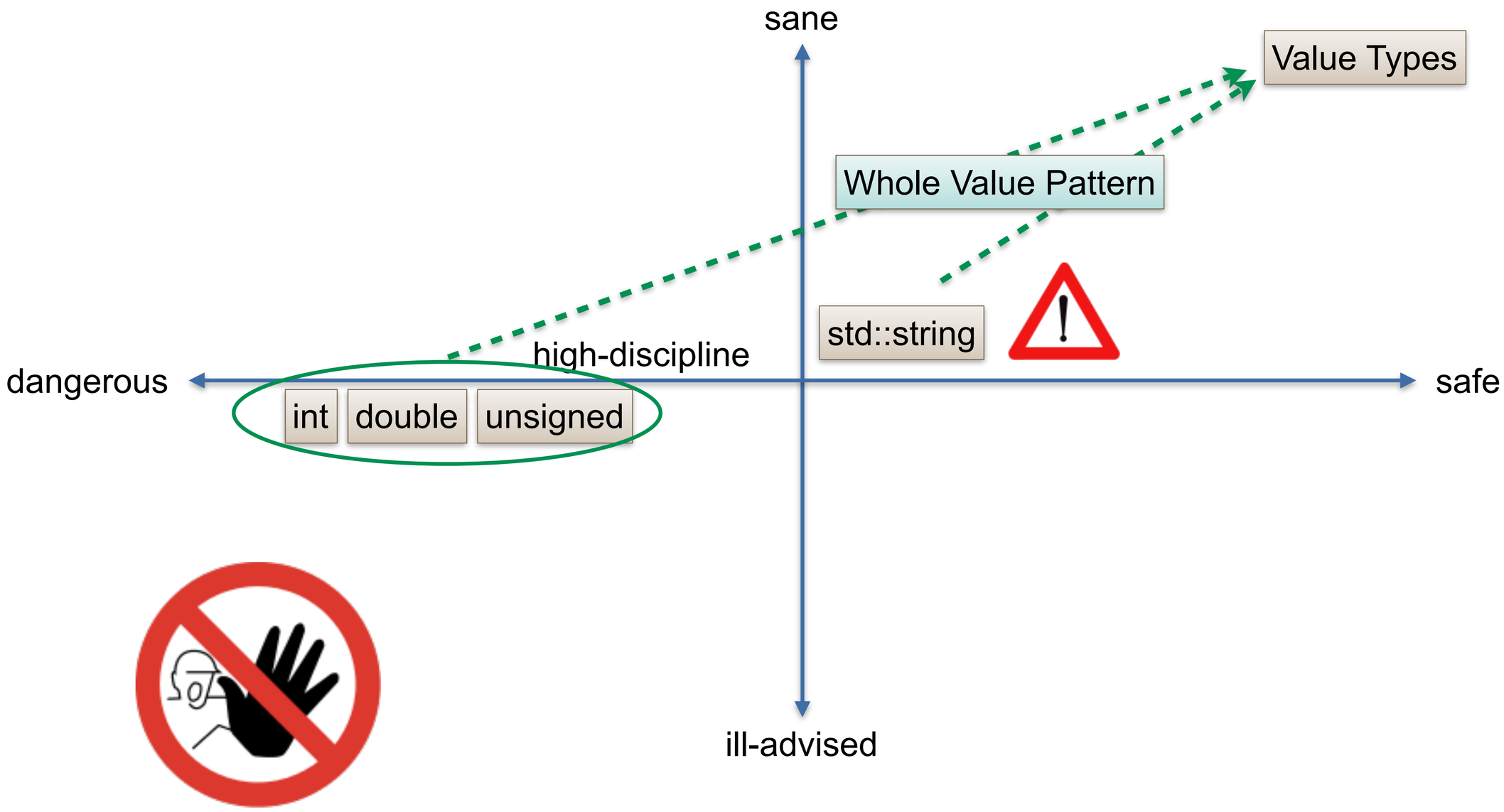
11010 .LX0: lib.f: .text // \s+ Intel Demangle

```
1  .LCPI0_0:
2          .quad   4636737291354636288     # double 100
3  consumption(double, double):                        # @consumption(
4          divsd   xmm1, qword ptr [rip + .LCPI0_0]
5          divsd   xmm0, xmm1
6          ret
```

Output (0/0) x86-64 clang 8.0.0 - 889ms (7877B)

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

Dimensions Safety and Sanity...



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

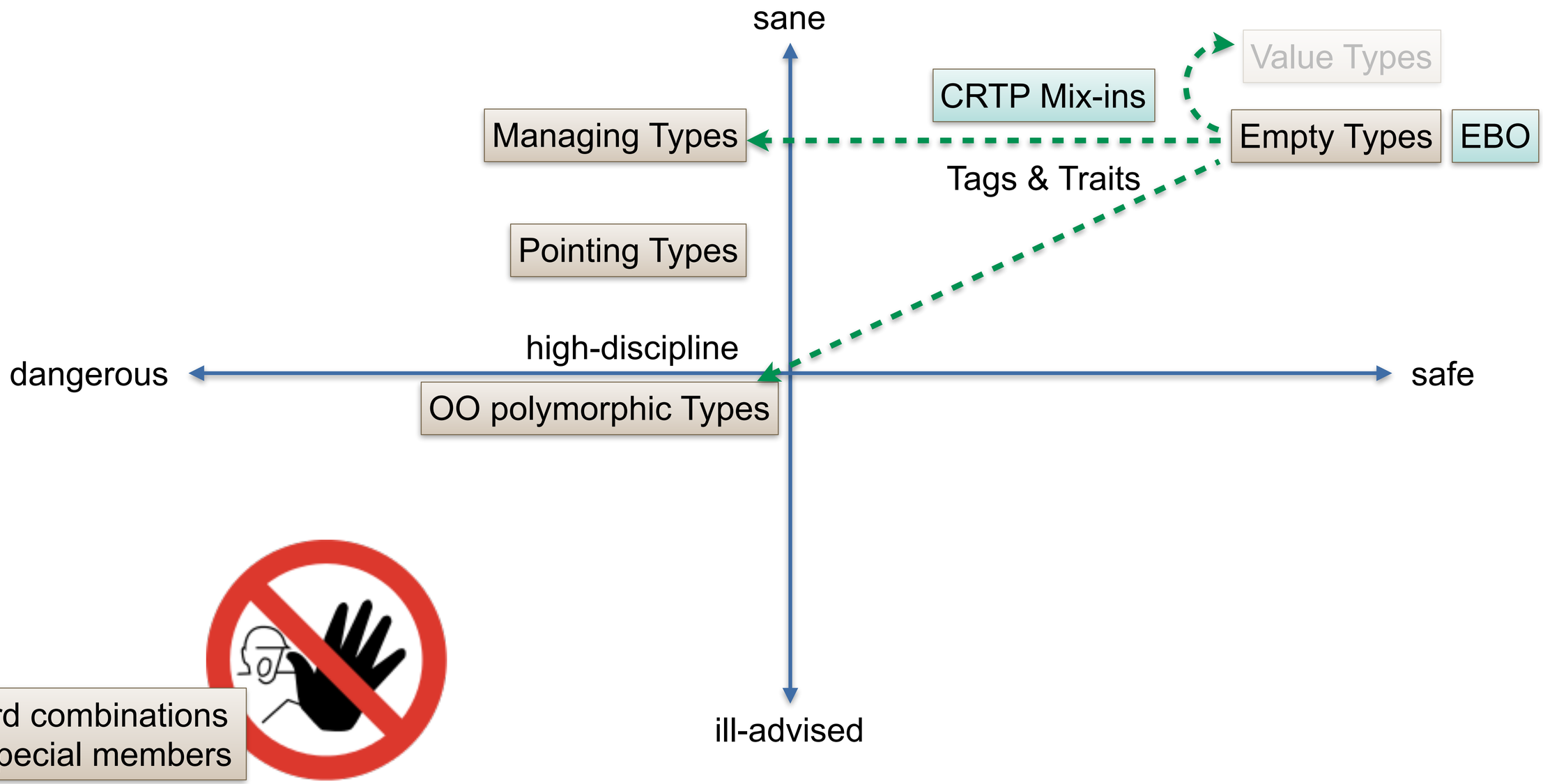
Empty Classes - useful?

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

"Something for Nothing" -- Kevlin Henney, 1999

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

Dimensions Safety and Sanity...



weird combinations
of special members

Peter's simple strong typing: PSSST empty base classes applied

- **CRTP base class**
  - provide value type and self
- **Operator-mix ins**
  - simplify specifying many with ops<>
  - Eq = Equality comparison
  - Out = generic operator>>
  - ScalarMult = km * factor
- **Only define operations needed**
  - no inadvertent computation
  - no unwanted conversions
- **consumption could be ctor**

```
using namespace Pssst;
struct literGas:strong<double,literGas>{};

struct literPer100km:strong<double,literPer100km>
,ops<literPer100km,Eq,Out>{};

struct kmDriven:strong<double,kmDriven>
,ScalarMult<kmDriven,double>{};

literPer100km consumption(literGas l, kmDriven km) {
  return {l.val/(km/100.0).val};
}

void testConsumption1over1(){
  literGas const l {1} ;
  kmDriven const km { 1 } ;
  ASSERT_EQUAL(literPer100km{100.0},consumption(l,km));
}
void testConsumption40over500(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(literPer100km{8.0},consumption(l,km));
}
```

- **Tag Types - overload resolution**
  - iterator_traits, in_place_t,
- **Traits - compile time programming**
  - querying and type composing traits
  - querying in static_assert to ensure no bad things happen
- **size optimization of template classes**
  - see unique_ptr

- **An optimal unique_ptr specialization for C-style pointers requiring free**

```
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;
}
```

code from CUTE

not allowed

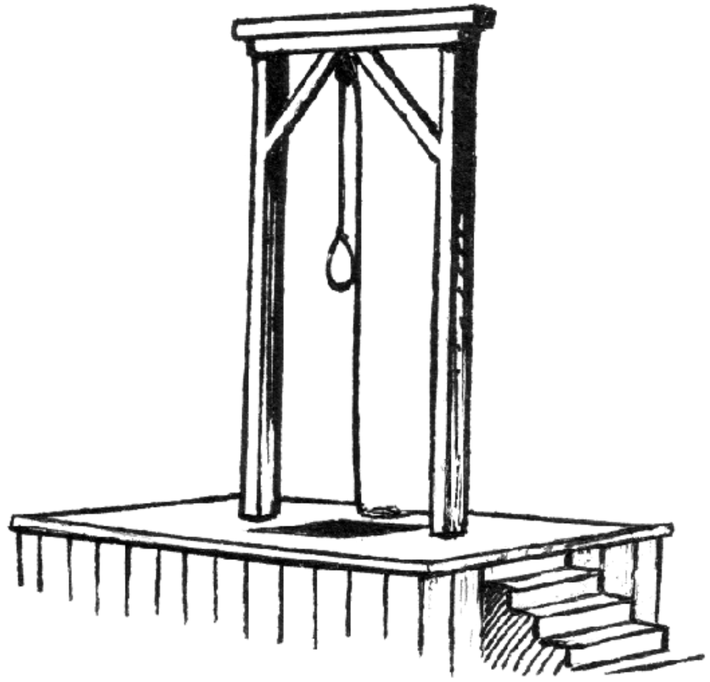
```
inline std::string plain_demangle(char const *name){
   if (!name) return "unknown";
   std::unique_ptr<char const, decltype(&std::free)>
   toBeFreed { abi::__cxa_demangle(name,0,0,0), &std::free};
   std::string result(toBeFreed?toBeFreed:name);
   return result;
}
```

```
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;
}
```

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

"C++ provides a rich set of types whose objects may dangle...We call these types potentially dangling. If a referred object's lifetime ends before the referring object, one risks undefined behavior."
(paraphrased from WG21-SG12/WG23 workshop in Kona 2019)

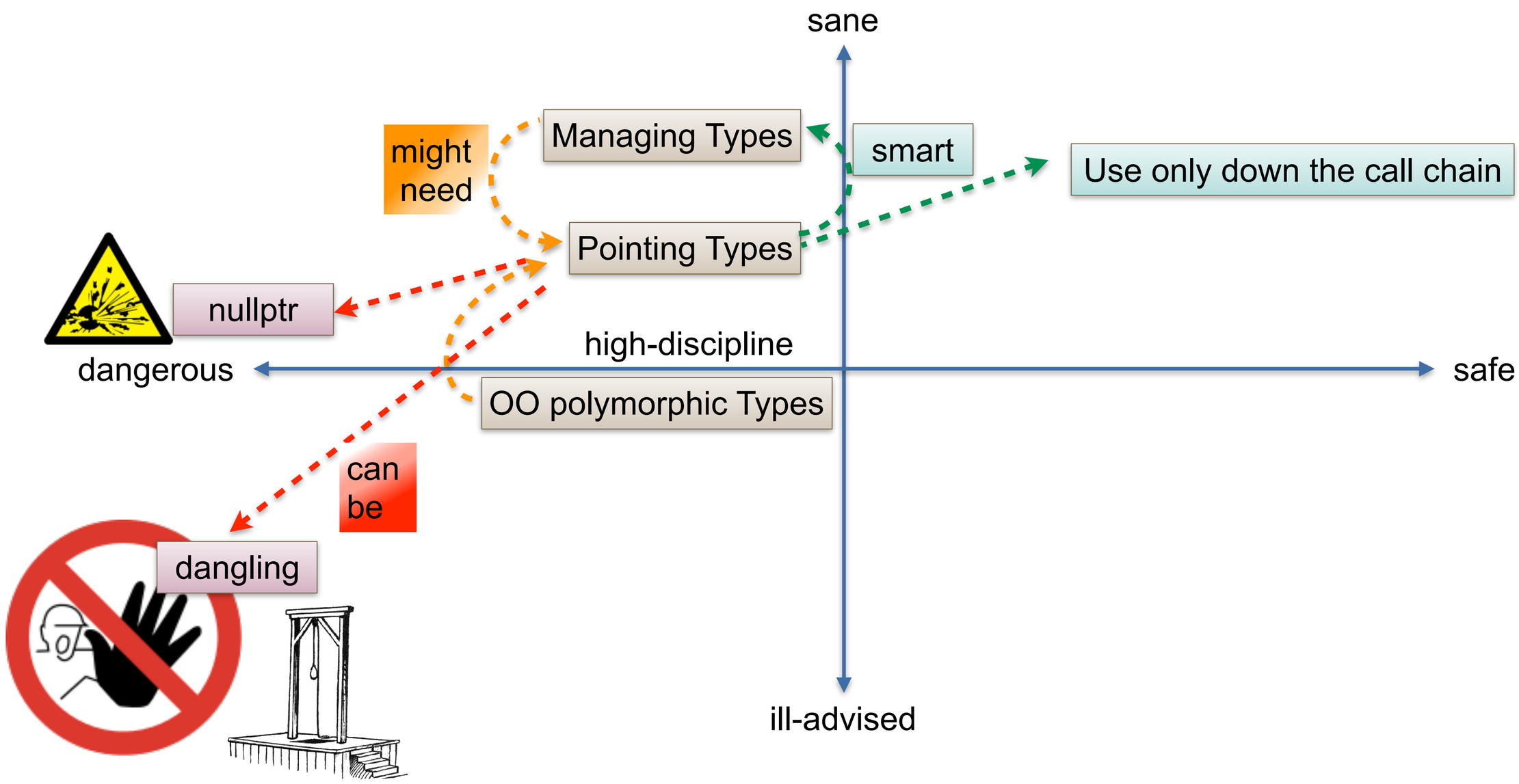
Cevelop C++
Your C++ deserves it



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

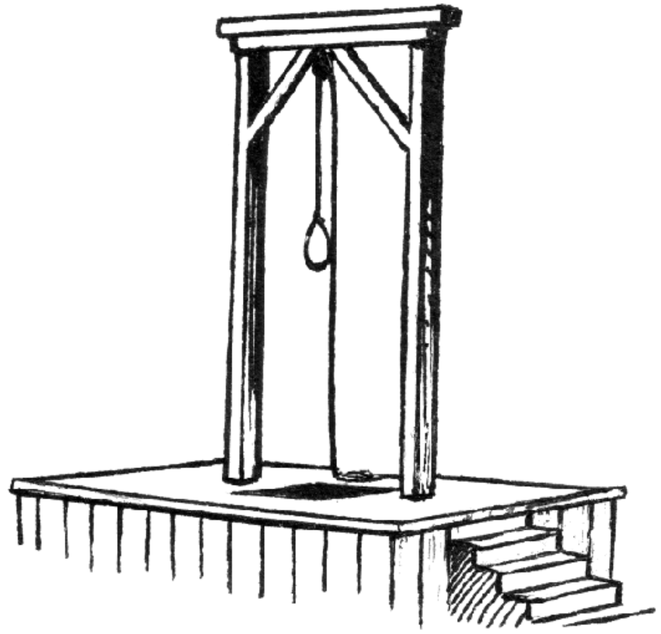
Dimensions Safety and Sanity... and "pointing" - potentially-dangling types



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

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 not return std::string_view unless you can trust your users and educate them!**

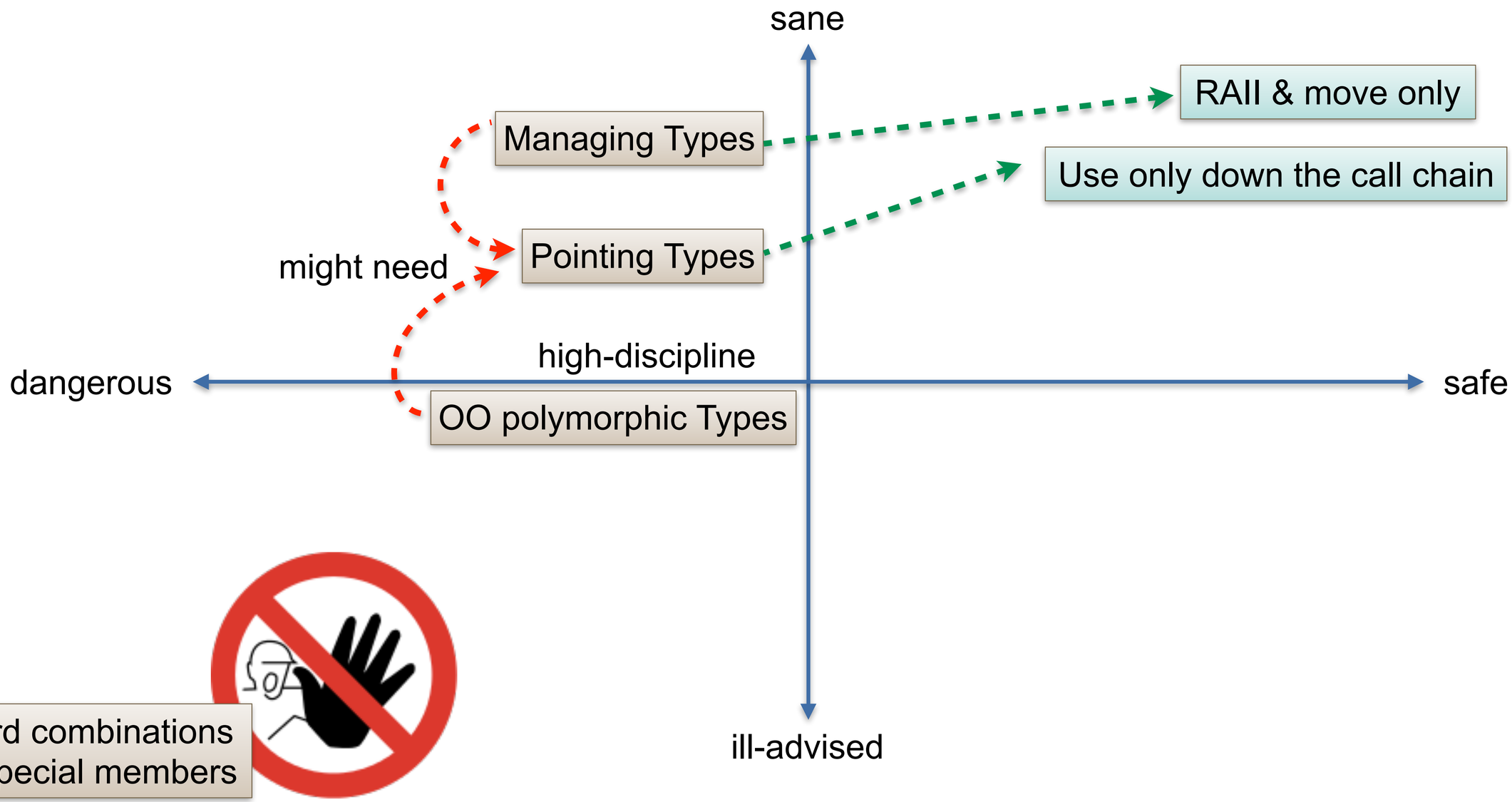
Managing stuff

"monomorphic object types"
-- Richard Corden, PRQA

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

Managing types have an
interesting **destructor**

Dimensions Safety and Sanity...



weird combinations
of special members

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.

Making a 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.

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

DesDeMovA

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

minimum amount of code to achieve desired non-copyable

What you write

	default constructor				move constructor	move assignment
nothing	defaulted			ulted	defaulted	defaulted
any constructor	not declared					
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.

- **Common to Managing types**
  - define "interesting" destructor: `~manager() { /* clean up stuff */}`
- **0: scope - 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)

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 (not in C++20)**~~
  - You can help: https://github.com/PeterSommerlad/scope17

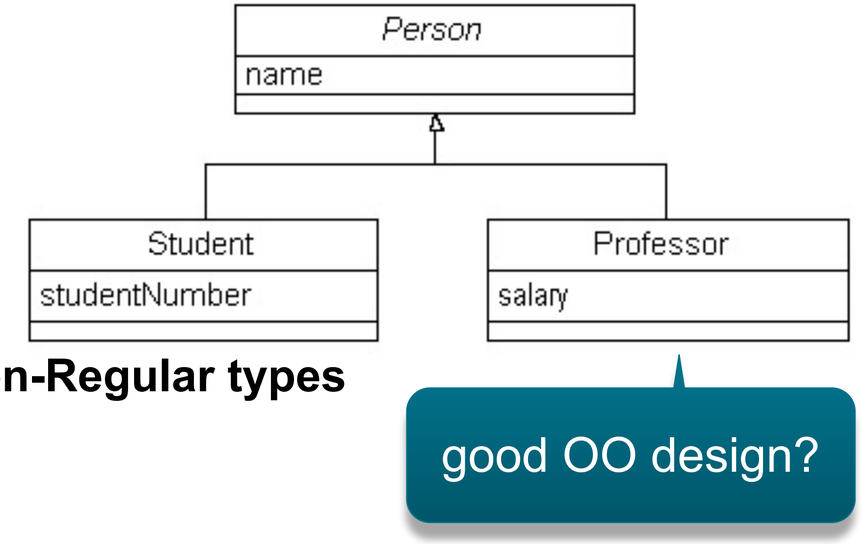
Dynamic Polymorphism

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

A polymorpic base class needs to define a virtual destructor, best as =default;

Polymorphic Object Types -- think thrice about using virtual!

- **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 require virtual destructor in base class**
  - subclasses should not add additional virtual members
  - Most other attempts with multiple layers of inheritance or even multiple inheritance are often futile
- **If set of Alternatives is closed, consider std::variant<T1,T2,T3> (or boost::variant)**
  - and visit(overloaded(...)) for dispatch



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

DesDeMovA

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

minimum amount of code to achieve desired non-copyable

What you write

	default constructor				move constructor	move assignment
nothing	defaulted			ulted	defaulted	defaulted
any constructor	not declared					
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.

We want Rule of Zero to Rule!

But what do we get, when we apply it?

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 / =default	defaulted	defaulted	defaulted	defaulted	defaulted
Scope Manager	typical	none / =default	implemented	deleted	deleted	deleted	=delete 🙋
Unique Manager	typical	defined / =default	implemented	deleted	deleted	implemented	implemented
Value Manager	yes	defined / =default	implemented	implemented	implemented	implemented	implemented
OO - Base	may be	may be	=default virtual!	deleted	deleted	deleted	=delete 🙋
OO & Value type erasure*	yes	no	Expert Level - =default	Expert Level Implementation	Expert Level Implementation	Expert Level Implementation	Expert Level Implementation

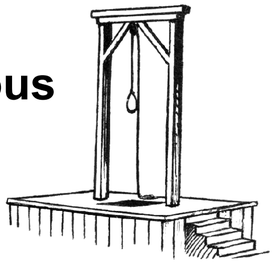
* see Sean Parent's talks and slides: https://sean-parent.stlab.cc/papers-and-presentations/#better-code-runtime-polymorphism

We want to apply the Rule of Zero
because
Code that is not there can not be wrong

Influence of Member Variable Types on Special Member Functions

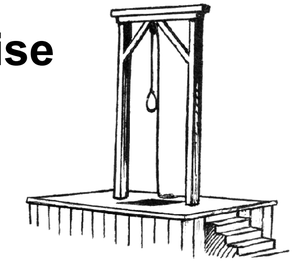
Member Varible Kind	Some constructor	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Value	none	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
T&[2]	yes	=delete	defaulted	defaulted	=delete[1]	defaulted	=delete[1]
Manager: [2] Scope	typical	none	defaulted	deleted	deleted	deleted	deleted 🤷
Manager: [2] Unique	typical	defined / =default	defaulted	deleted	deleted	defaulted	defaulted 🤷
Pot. Dangling[2,3]	typical	defined / =default	defaulted	defaulted	defaulted	defaulted	defaulted

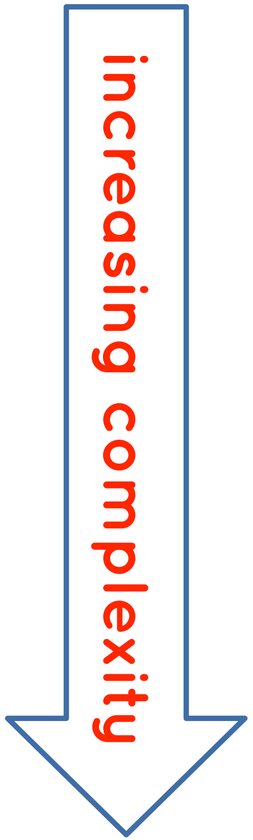
- **1 - remedy through using std::reference_wrapper<T> instead**
- **2 - "contagious": your class becomes the same without further means**
- **3 - Regular Potentially Dangling Members make using your class type dangerous**



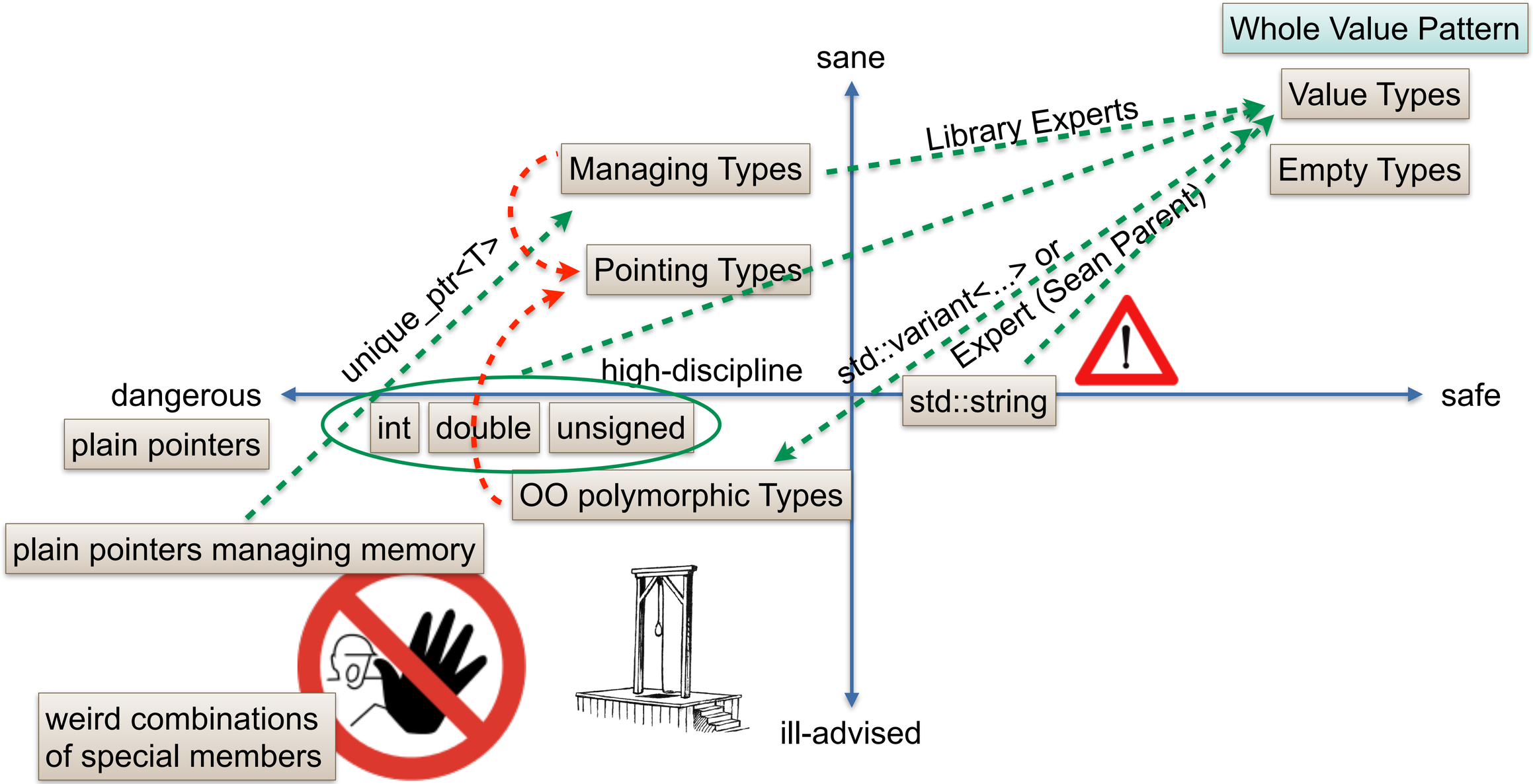
- **Rule of Zero**
  - for value types, for types with managing members
- **Rule of DesDeMovA**
  - for OO base classes, for SBRM classes

- **Adapted Rule of Three (destructor and move operations)**
  - for unique managing types define move operations, think of a sane moved-from state
- **Rule of Five**
  - for expert-level managing types (Containers like vector, Type Erasure, others)
- **Avoid members of potentially dangling types, otherwise**





Dimensions Safety and Sanity...



- **Model with Value Types almost always**
- **Wrap primitives using Whole Value, even a named simple struct communicates better than** int (see my other talk, just a reminder!)
- **Be aware of the required expertise and discipline for Manager types and OO hierarchies**
  - Remember "Rule of DesDeMovA" 🤚
  - **and make yourself and your environment familiar with it**
- **Be very disciplined about using Pointing types, this includes references and string_view**
  - Member variables that can potentially dangle make your type potentially dangling as well!
- **Run away from types with weird special member function combinations, even if defaulted**
  - usually they attempt to do too much or the wrong thing -> REFACTOR!


Advanced
Developers
Conference
++
Development for Professionals!

Fragen?

Vielen Dank!

Ich freue mich auf Feedback!

Peter Sommerlad