C-Minus-Minus
Instead of C-Plus-Plus?

Efficient C++ for Embedded Programming me

– Know **WHAT** to use –

– Know what **NOT** to use –

Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
http://tbfe.de

FPGA-Kongress 2016
Sprachen / Languages
2016-07-13 (09:45 ... 10:3

Agenda

1. C++ as Superset of C
2. What You May Use Blindly
3. What You Should Use Judiciously
4. What You Might Want to Avoid

You are welcome to interrupt the speaker with questions* and – es
during the code-walk –– propose to try small changes.

*: Your questions will be answered during the presentation to the best of the speaker's abilitie
in private communication after the presentation.

C++ as Superset of C

This means:

- Everything that can be done in C ...
- ... can be done in C++ too*
- **AND MUCH MUCH MORE**

So the art of using C++ in Embedded Projects is to

- use only what is beneficial and
- leave out everything else.

*: Well, not quite: some of the "bad parts" which are still in the C language have been cut off of this is still present in C for historical reasons and to keep the language backward comp software base that still makes use of these features. The pressure was not quite that stro the principle could be adopted: *As close to C as possible, but no closer.*

What You May Use Blindly

- Name Spaces
- Function Overloading
- C++-Style Casts
- Classic References
- Access Protection for `structs`
- STL Algorithms (for Native Arrays)
- C++11/14/1z ...
  - ... `nullptr`
  - ... `constexpr`
  - ... Scoped `enums`
  - ... Type Aliases
  - ... `auto`-Variables
  - ... Uniform Initialisation (... and maybe even more ...)

Name Spaces

To organize code via the C++ `namespace`-s is

- a pure compile-time feature
- an improvement over lexically qualified names

See also:
http://en.cppreference.com/w/cpp/language/namespace

Function Overloading

Overloading is

- a pure compile-time feature
- an improvement over having to **use and remember*
  - different function names for
  - semantically equivalent operations

See also:

http://en.cppreference.com/w/cpp/language/overload_resolution

C++-Style Casts

The new cast syntax introduced with C++98

- better expresses the purpose of a type conversion
- avoids spurious errors on the developer's side

See also:
http://en.cppreference.com/w/cpp/language/static_cast
http://en.cppreference.com/w/cpp/language/dynamic_ca
http://en.cppreference.com/w/cpp/language/const_cast
http://en.cppreference.com/w/cpp/language/reinterpret_cast

Classic References

Classic references are

- at the **low-level view** a different syntax for pointers
- at the **high-level view** aliases names for memory locations

See also:
http://en.cppreference.com/w/cpp/language/reference

Access Protection for `struct`s

Compared to "opaque types" implemented via C-`structs`

- C++ classes provide better access protection
- with **no overhead at run-time** introduced at run-time

Designing opaque types **does not mean to**

- to simply make all data members `private` and
- provide `public` *accessors* and *modifiers* at a "1:1" base

Instead **which operations to provide** should

- consider the typical client's use and
- especially strive to protect invariants

STL Algorithms (for Native Arrays)

All the algorithms provide by the STL

- can be applied to native array
- by using native pointers as iterators

See also: http://en.cppreference.com/w/cpp/algorithm
http://en.cppreference.com/w/cpp/numeric

C++11 `nullptr`

Via the keyword `nullptr`

- C++11 provides a unique value of type `std::nullptr_t`
- **convertible to any pointer type**
- **not comparing equal** to any valid memory address

See also: http://en.cppreference.com/w/cpp/language/nu

C++11 `constexpr`

The `constexpr` qualifier applied to

- variables guarantees compile-time initialisation
- functions allows their call in compile-time initialisation of varia

See also:
http://en.cppreference.com/w/cpp/language/constexpr

C++11 Scoped enums

Scoped enums

- cause no pollution of their surrounding name space
- without the requirement for lexical qualification

See also:
http://en.cppreference.com/w/cpp/language/scoped_enu

C++11 Type Aliases

Type aliases provide a "left-to-right" style writing type definitions:

Classic type definitions:

```
typedef unsigned long long ULL;
```

C++11 type aliases:

```
using ULL = unsigned lon
```

See also:
http://en.cppreference.com/w/cpp/language/type_alias

C++11 `auto`-Variables

By using `auto` for the type of a variable the actual type

- is **statically** deduced from the initialisation expression
- (in a way that most often* meets the expectations of the user)

See also: http://en.cppreference.com/w/cpp/language/au

*: But not always ... there are some border cases with respect to initializer lists which may cau

C++11 Uniform Initialisation

See also:
http://en.cppreference.com/w/cpp/language/value_initia
http://en.cppreference.com/w/cpp/language/direct_initia
http://en.cppreference.com/w/cpp/language/copy_initialization
http://en.cppreference.com/w/cpp/language/list_initialization
http://en.cppreference.com/w/cpp/language/aggregate_initializat
http://en.cppreference.com/w/cpp/language/reference_initializati

What You Should Use Judiciously

- Inline Functions
- Operator Overloading
- Class Specific Type Conversions
- Composition and Inheritance
- Runtime Polymorphism
- General Runtime Type-Identification
- "GoF-Style" Design Patterns
- Templates for ...
  - ... Type Generic Classes
  - ... Type Generic Algorithms
  - ... Configurable Policies
  - ... Meta-Programming
- C++11 ...
  - ... Rvalue References
  - ... Literal Suffixes
  - ... Lambdas
  - ... `std::function` and `std::bind`
  - ... Smart Pointers
- Extensions from the Boost Platform

Inline Functions

Using the `inline` qualifier on a function

- will trade
  - the memory foot-print (with respect to code space)
  - for run-time performance (and better code linearity)
- **may be a "win-win" situation in case or very small funct**

See also: http://en.cppreference.com/w/cpp/language/in

Operator Overloading

Overloading **may** improve readability of some source code

- **if** it follows the traditional meaning of the operator symbol
- **may** invoke expectations on the client's side that cause a lot o

See also:
http://en.cppreference.com/w/cpp/language/operators
http://www.boost.org/doc/libs/release/libs/utility/operato

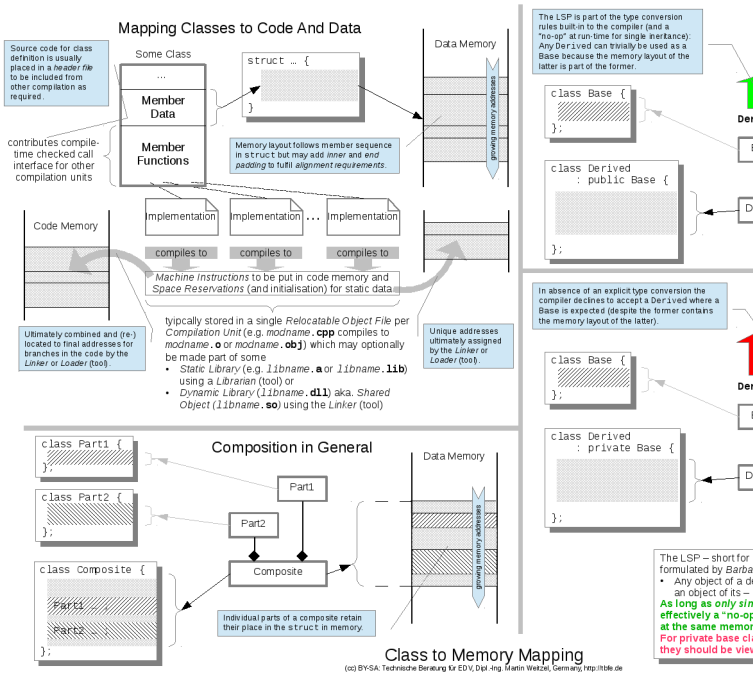
Class Specific Type Conversions

Automatic conversions may be defined by

- a constructor describing
  - how a different type is
  - converted into an object of its class
- a type-case operation describing
  - how an object of its class is
  - converted into a different type

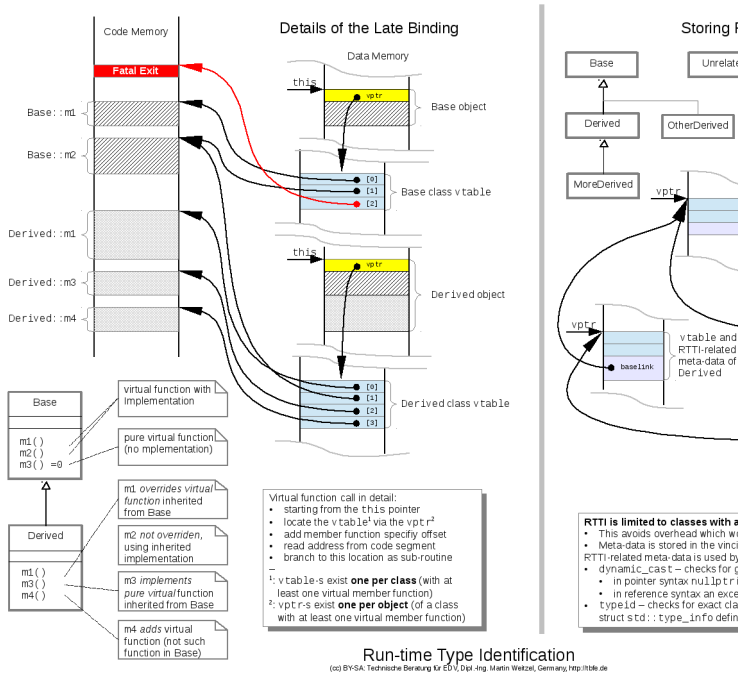
(i) See also:
http://en.cppreference.com/w/cpp/language/copy_initiali
http://en.cppreference.com/w/cpp/language/cast_operat
http://en.cppreference.com/w/cpp/language/explicit

Composition and Inheritance



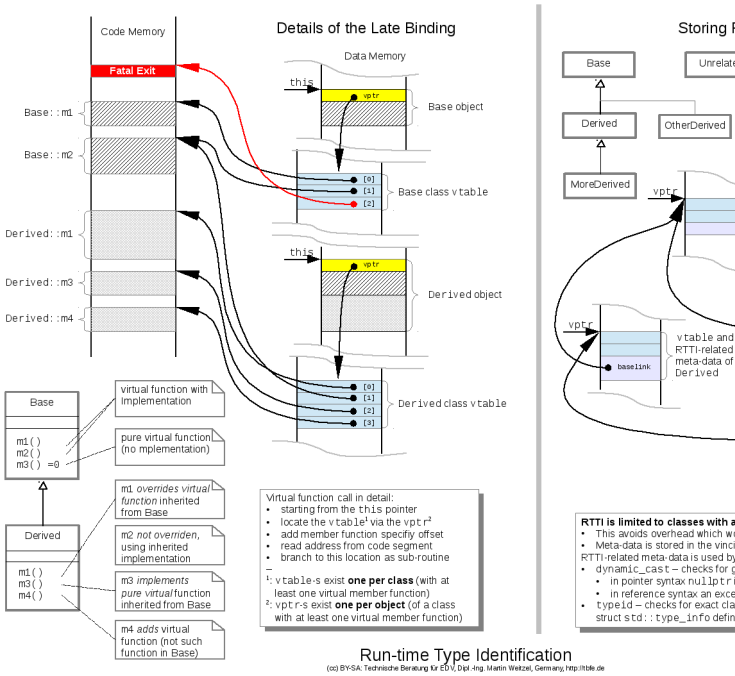
Runtime Polymorphism

(see left part of graphic)



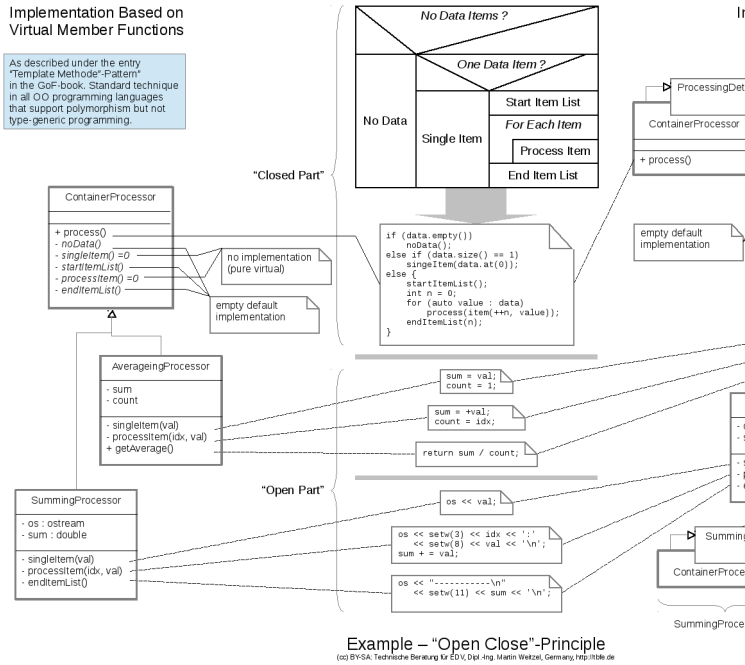
General Runtime Type-Identification

(see right part of graphic)



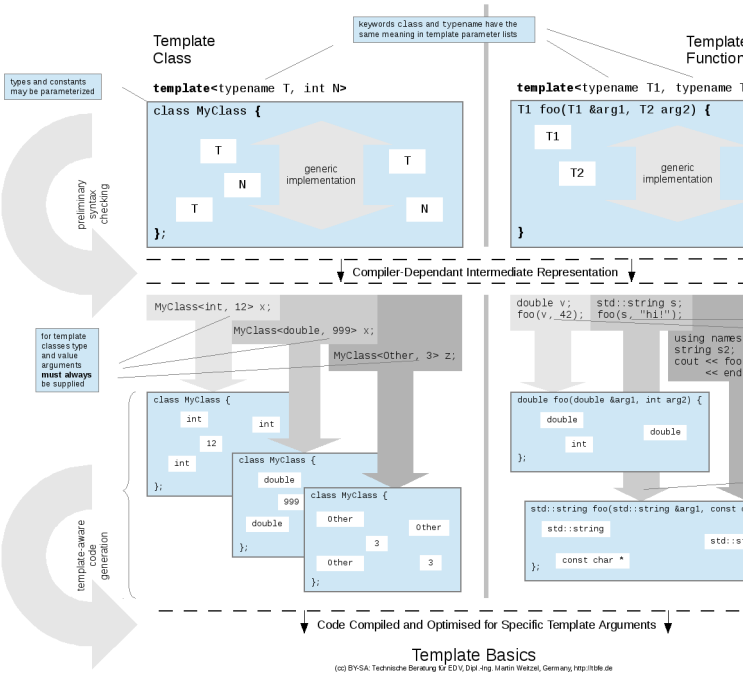
"GoF-Style" Design Patterns

(see left part of graphic)



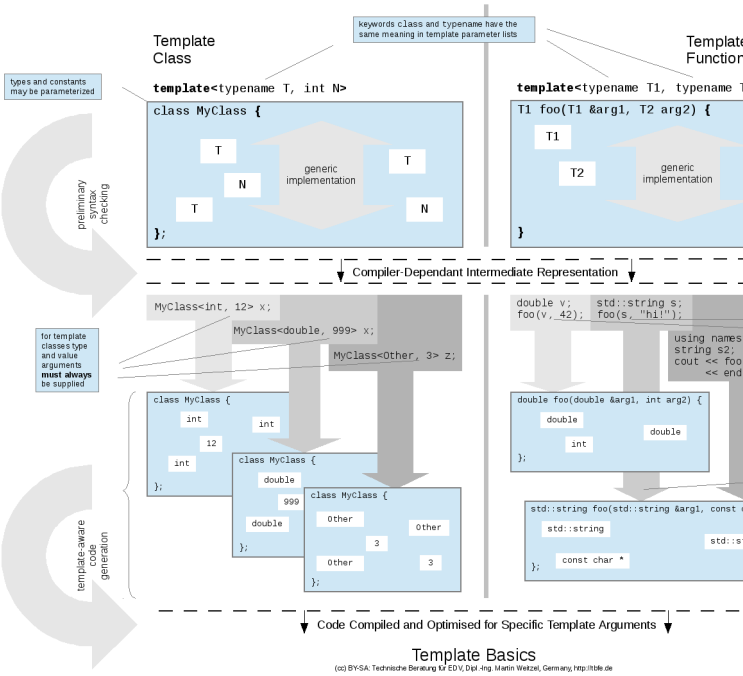
Type Generic Classes

(see left part of graphic)



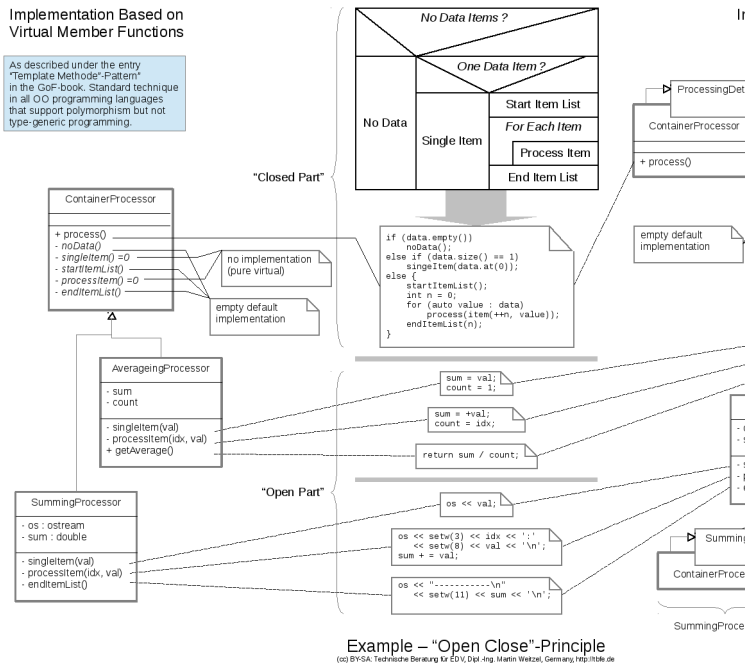
Type Generic Algorithms

(see right part of graphic)



Template Based Policies

(see right part of graphic)



Template Meta-Programming

Meta programming means to use templates as compile-time funct

- transforming some "input"
  - consisting of (concrete) types or
  - values that can be calculated at compile-time
- into some "output"
  - consisting of a (concrete) type or
  - values that is determined compile-time

> Meta programming applies in a lot of areas
> which may not seem obvious at a cursory look ...*

*: Otherwise, sorry that barrel is too large to be opened here quickly ...

C++11 Rvalue References

Rvalues are values

- which have some storage
- but (very) limited life-time

http://en.cppreference.com/w/cpp/language/reference

C++11 Literal Suffixes

User defined literal suffixes provide a similar notation as used to d
physical units.

See also:
http://en.cppreference.com/w/cpp/language/user_literal

C++11 Lambdas

Lambdas – also known as function literals – are

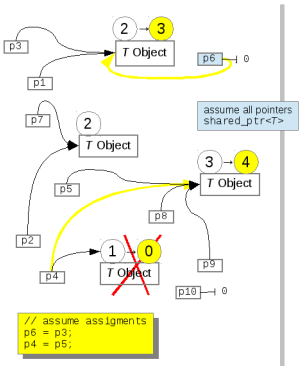
- especially useful to control some aspect of an STL algorithms
- compete with the use of *Functors* and *Function Pointers*

See also: http://en.cppreference.com/w/cpp/language/la

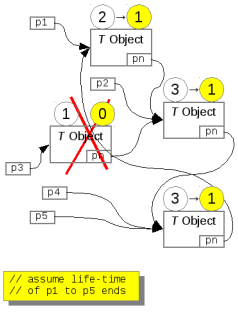
C++11 `std::function` and `std::bind`

See also:
http://en.cppreference.com/w/cpp/utility/functional/func
http://en.cppreference.com/w/cpp/utility/functional/bind

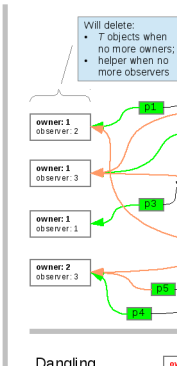
C++11 Smart Pointers



Reference Counting Principle



Problem of Cyclic References



Dangling Weak Pointer

Comparing ...	`std::unique_ptr<T>`	`std::shared_ptr<T>`	Remarks
Characteristic	refers to a single object of type *T*, **uniquely owned**	refers to a single object of type *T*, **possibly shared with other referrers**	may also refer to "no object" (like a `nullptr`)
Data Size	same as plain pointer	same as a plain pointer plus some extra space per referred-to object	
Copy Constructor	no*	yes	particularly efficient as only pointers are involved
Move Assignment	yes		
Copy Assignment	no*		a *T* destructor must also be called in an assignment if the current referrer is the only one referring to the object
Move Assignment	yes		
Destructor (when referrer life-time ends)	always called for referred-to object	called for referred-to object when referrer is the last (and only) one	

*: explcit use of `std::move` for argument is possible

Smart Pointer Comparison

C++11 Smart Pointers (2)

See also:
http://en.cppreference.com/w/cpp/memory/unique_ptr
http://en.cppreference.com/w/cpp/memory/shared_ptr
http://en.cppreference.com/w/cpp/memory/weak_ptr

Extensions from the Boost Platform

See also: http://www.boost.org

What You Might Want to Avoid

GENERALLY:

- Everything not Properly Understood
- Badly Structured Libraries

SPECIFICALLY:

- `virtual` Destructors
- Diamond-Shaped Inheritance
- Compiling with Exceptions Enabled

Everything not Properly Understood

> Don't play with a scalpel knife
> until you have a good understanding
> how easy and deep it cuts.

Badly Structured Libraries

Many Libraries have been designed with a "workstation/server"-ty
hardware in mind

- where memory comes at (seemingly) "no cost" and
- everything is loaded as shared object / DLL anyway

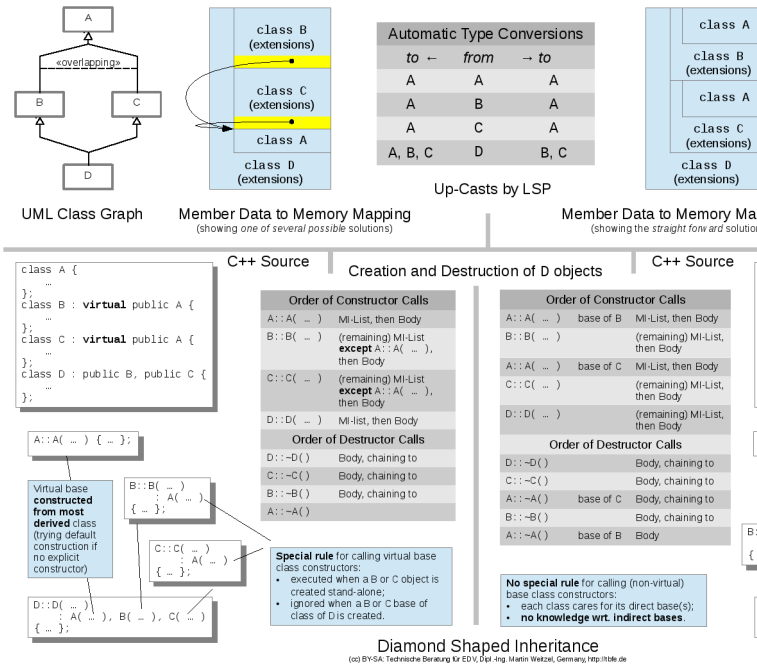
> Such libraries often require a **major restructuring** to make ther
> usable for static linking at a "fine grained" resolution.

`virtual` Destructors

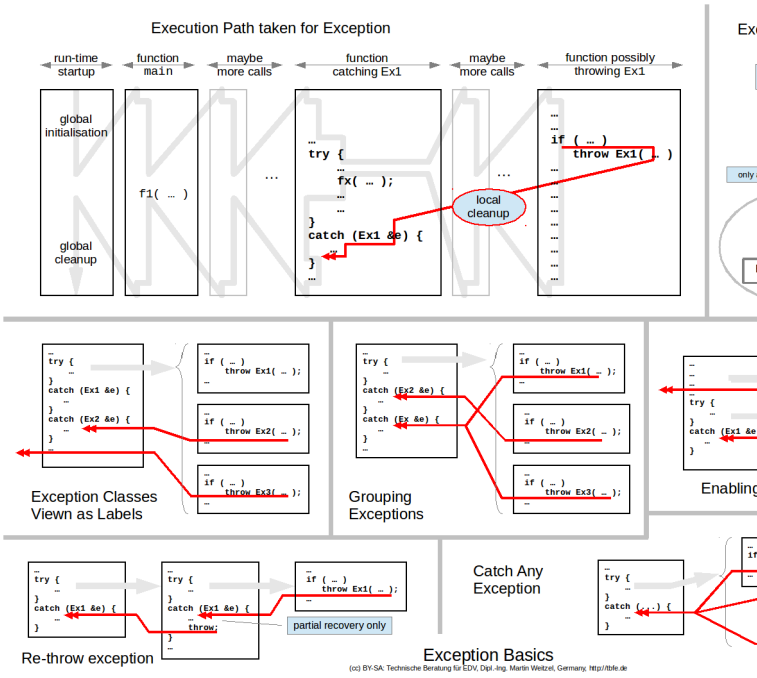
Be sure to understand that the "requirement" to use `virtual` destr
always if a class is a base class from which other classes are deri

- **may** have a substantial negative impact on run-time performa
- may add some a good degree of robustness to the code base
- **but is really necessary only** if derived classes are
  - allocated on the heap and later
  - destroyed via base-class pointers

Diamond-Shaped Inheritance



Compiling with Exceptions Enabled



And More – If Time Allows

So you name it ...

... otherwise:

Thanks for Listening!