

Top Left

Top

Top Right

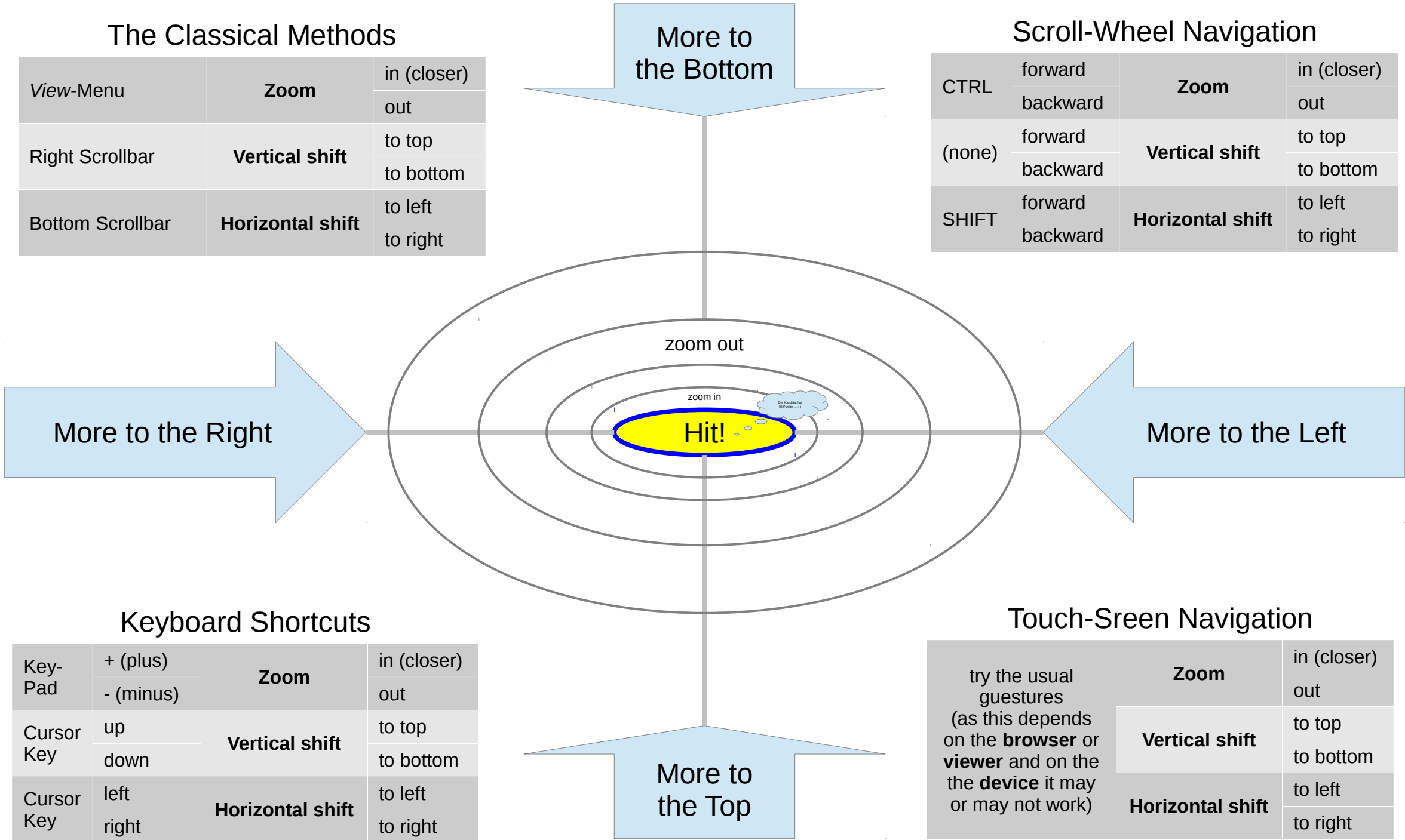
Enjoy the White Nights

LEFT

Wild Wild West

The Classical Methods			
View-Menu	Zoom	in (closer) out	
Right Scrollbar	Vertical shift	to top to bottom	
Bottom Scrollbar	Horizontal shift	to left to right	

Scroll-Wheel Navigation			
CTRL	forward backward	Zoom	in (closer) out
(none)	forward backward	Vertical shift	to top to bottom
SHIFT	forward backward	Horizontal shift	to left to right



Eastern Wisdom

RIGHT

Keyboard Shortcuts			
Key-Pad	+ (plus) - (minus)	Zoom	in (closer) out
Cursor Key	up down	Vertical shift	to top to bottom
Cursor Key	left right	Horizontal shift	to left to right

Touch-Sreen Navigation		
try the usual guestures (as this depends on the browser or viewer and on the the device it may or may not work)	Zoom	in (closer) out
	Vertical shift	to top to bottom
	Horizontal shift	to left to right

Bottom Left

Beware of the Penguins

Bottom Right

Test Browser Navigation

C-Compatibility

```
std::string filename;  
...  
FILE *fp = fopen(filename.c_str(), "r")
```

Where a C-Style string (`const char *`) is expected an `std::string` must be explicitly converted ...

`CharType`
`std::basic_string`

Lookup in reference documentation [here](#) ...

`char`
`std::basic_string`

... but prefer these typedef-s for readability!

`wchar_t`
`std::basic_string`

`std::string`

`std::wstring`

```
void foo(const std::string &);  
...  
int main() {  
    foo("hello, world");  
}
```

... the other way round is automatically

```
char ch;  
std::string s  
while (get_next(c)) {  
    ...  
    s.append(&ch, 1);  
}
```

When accepting an `std::string` as read-only argument a const-reference should be used ...

```
void bar(std::string in);
```

... as for value arguments an (avoidable) copy would be created.

Algorithmically filling an `std::string` by always adding to its end can be considered efficient as reservations internally care for extra space.

Efficiency

hi!

a

Hello, world

```
std::string a("hi!");  
std::string b("Hello, world");
```

a = b;

becomes (silently) **shared**

a

Hello, world

b

content copied on write ...

a[7] = 'W';

... and again **unshared**

a

Hello, World

May or May Not Have COW-Implementation

Classes

Input and Output

For input

- use operator<> to skip leading white-space first, then read-in characters up to next white-space;
- use `std::getline` to read until given separator ('`\n`' by default).

For output

- operator<< has the usual behaviour.

```
int n = 0;  
...  
cout << ++n  
      << line  
      << endl;
```

```
// read standard input  
// line by line:  
using namespace std;  
string line;  
while (getline(cin, line)) {  
    ...  
}
```

convert to every builtin **integral** type

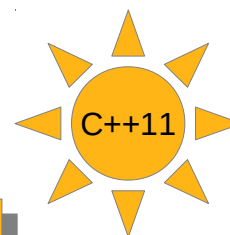
convert to every builtin **floating point** type

overloaded for every builtin **integral** and **floating point** type

```
std::string std::to_string( ... );
```

Numeric Conversions

Standard Strings may – more or less – be used like builtin types.



```
std::string s;  
...  
... std::stoi(s) ...  
... std::stol(s) ...  
... std::stoul(s) ...  
... std::stoll(s) ...  
... std::stoull(s) ...  
... std::stof(s) ...  
... std::stod(s) ...  
... std::stold(s) ...  
...
```

```
std::string str;  
...  
for (char c : str) {  
    // process str  
    // char-by-char  
}
```

Advanced operations:

- too many to list (→ RTFM).

Process like an STL container:

- the usual iterator subclasses are defined as for containers;
- therefore all STL algorithms may be used on `std::string` too.

C++11 has added some “missing” ones:

- e.g. get rid of over-allocation with `shrink_to_fit`

Basic String Operations

Similar to builtin types:

- construction, assignment, including comparison etc. ... (as can be expected);

Similar to builtin arrays:

- single character access via operator[];
- consider to use `at()` for index-checking at run-time;
- both return a reference, so assignment works too.

As in some other programming languages:

- concatenation with operator+;
- operator+= for combined assignment.

Basic searching:

- `find`, `rfind`, `find_first_of`, ...

Partition:

- `copy` (partially), `substr`, ...

Standard Strings 101

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

Advanced String Operations

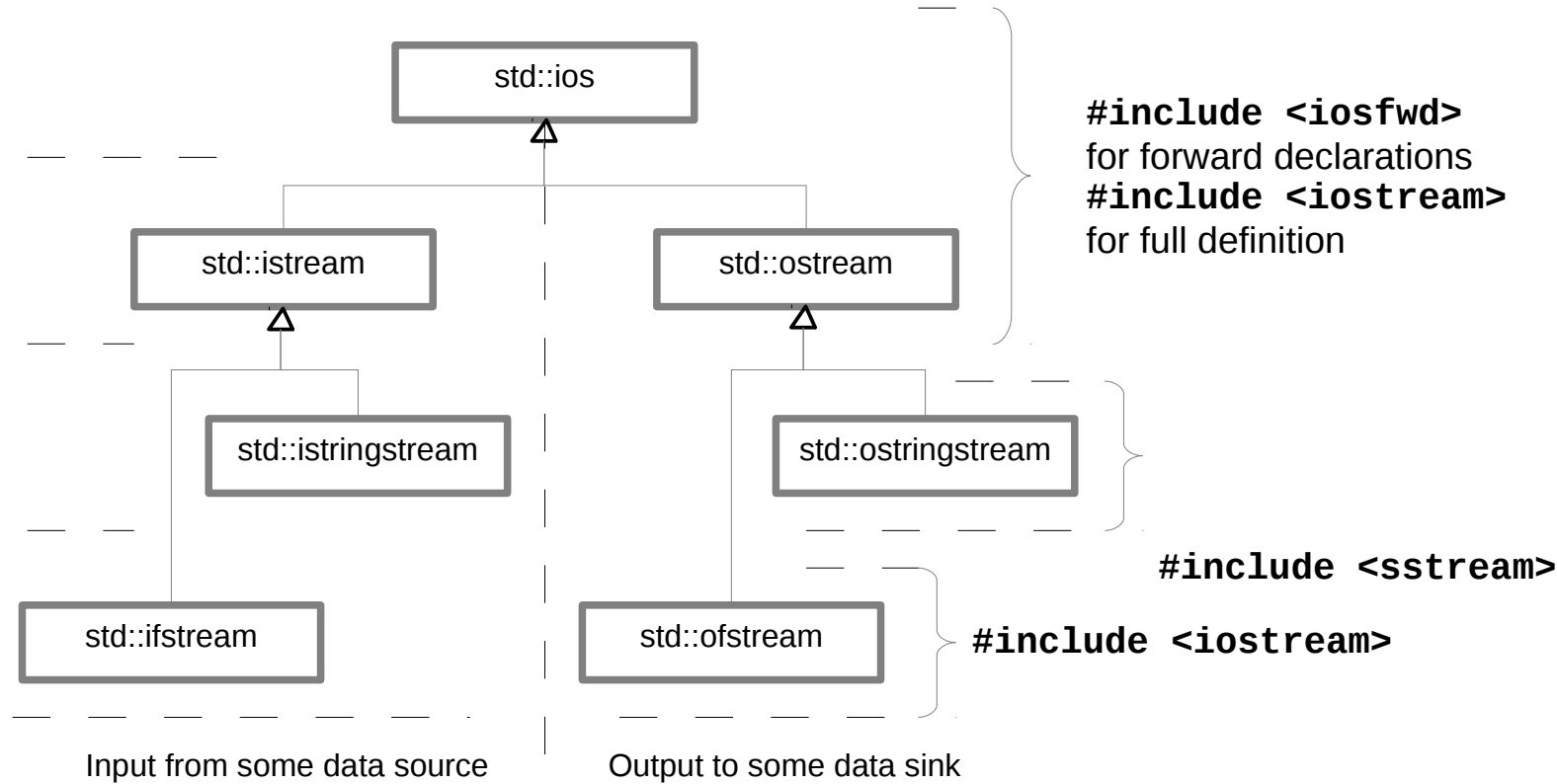
I/O-Streams Front-End

Common type definitions, constants, etc.

I/O-Operations are defined here – useful as function arguments

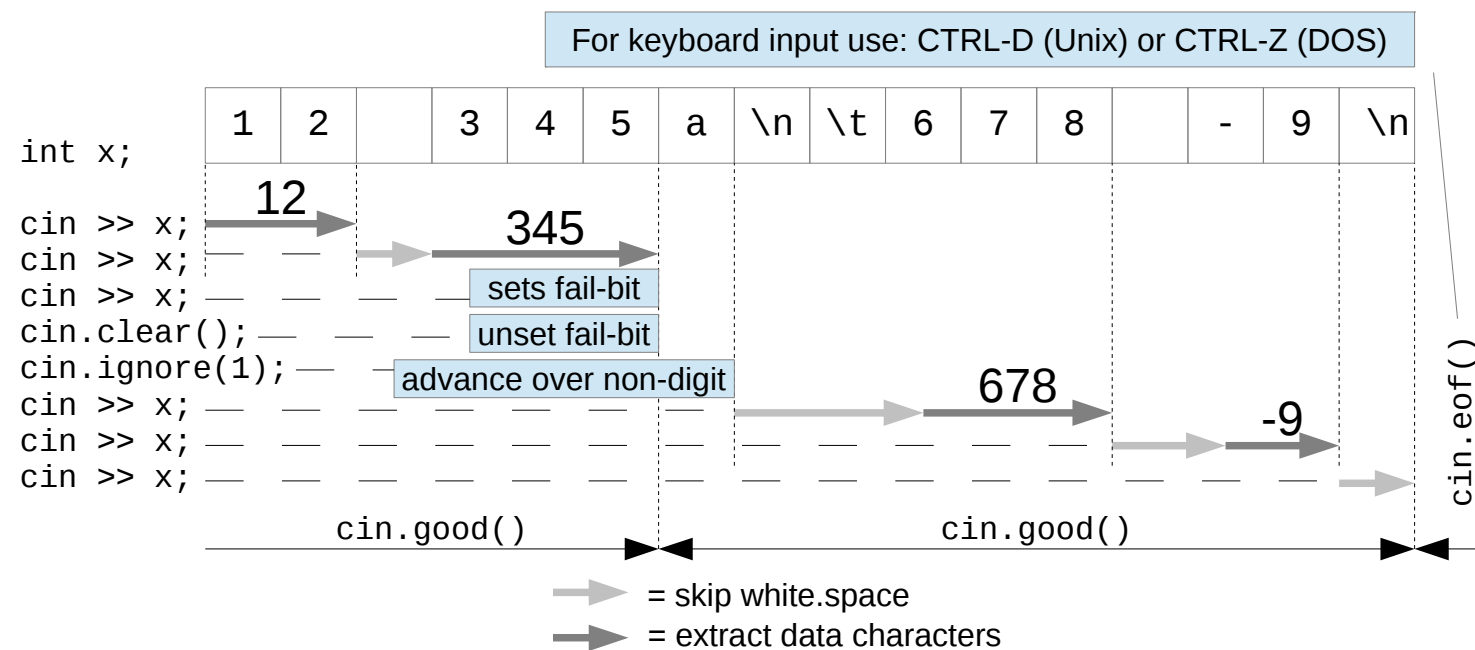
“I/O” taking place in memory of type `std::string`

I/O to/from external sources and sink (typically classic files or devices)



I/O-Stream States (assuming namespace `std` and stream named `s`)

Set ...	Name	is set ?	set explicitly	all unset ?	unset all
... on end of input	<code>ios::failbit</code>	<code>s.fail()</code>	<code>s.clear(ios::failbit)</code>		
... on format error	<code>ios::eofbit</code>	<code>s.eof()</code>	<code>s.clear(ios::eofbit)</code>	<code>s.good()</code>	<code>s.clear()</code>
(implem. defined)	<code>ios::badbit</code>	<code>s.bad()</code>	<code>s.clear(ios::badbit)</code>		

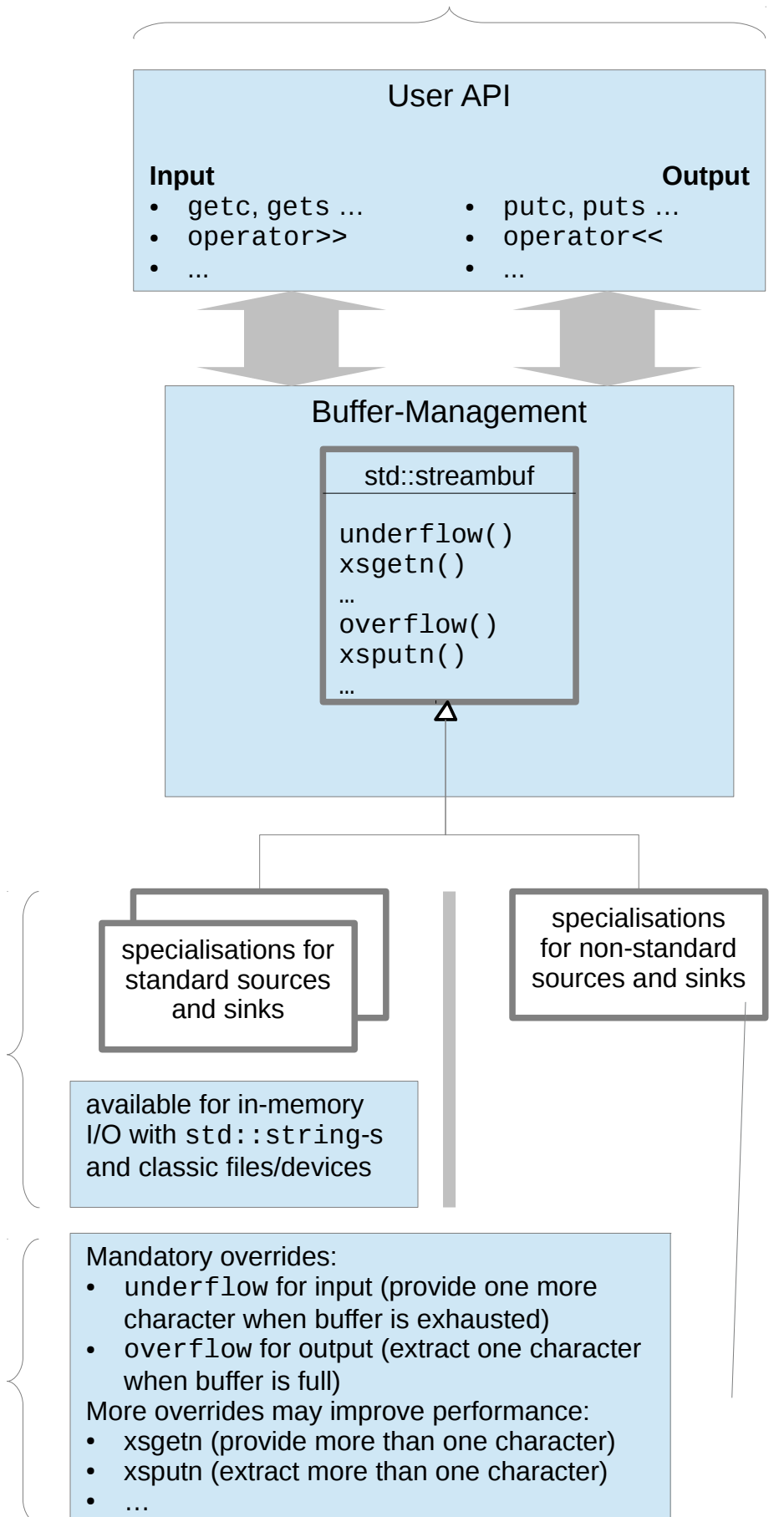


I/O-Streams State-Bits

I/O-Stream Basics

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

“day to day” use of C++



I/O-Streams Back-End

Parametrizing Type (double → T) and Size (11 → N+1)

```
class RingBuffer {
    double data[11];
protected:
    std::size_t iput;
    std::size_t iget;
    static std::size_t wrap(std::size_t idx) {
        return idx % 11;
    }
public:
    RingBuffer()
        : iput(0), iget(0)
    {}
    bool empty() const {
        return (iput == iget);
    }
    bool full() const {
        return (wrap(iput+1) == iget);
    }
    std::size_t size() const {
        return (iput >= iget)
            ? iput - iget
            : iput + 11 - iget;
    }
    void put(const double &);
    void get(double &);
    double peek(std::size_t) const;
};

void RingBuffer::put(const double &e) {
    if (full())
        iget = wrap(iget+1);
    assert(!full());
    data[iput] = e;
    iput = wrap(iput+1);
}

void RingBuffer::get(double &e) {
    assert(!empty());
    e = data[iget];
    iget = wrap(iget+1);
}

double RingBuffer::peek(std::size_t offset = 0) const {
    assert(offset < size());
    return data[wrap(idx + offset)];
}
```

Parametrizing Type

```
template<typename Type>
class RingBuffer {
    Type data[11];
    ...
    void put(const Type &);
    void get(Type &);
    Type peek(std::size_t) const;
};

template<typename Type>
void RingBuffer<Type>::put(const Type &e) {
    ...
}

template<typename Type>
void RingBuffer<Type>::get(Type &e) {
    ...
}

template<typename Type>
Type RingBuffer<Type>::peek(std::size_t offset = 0) const {
    ...
}
```

RingBuffer<double> b;

It makes sense to use the net-size here as leaving the last slot empty to differ between an empty and a full buffer can be considered to be an implementation detail.

```
template<std::size_t Size>
class RingBuffer {
    double data[Size+1];
    ...
    std::size_t size() const {
        return (iput >= iget)
            ? iput - iget
            : iput + (Size+1) - iget;
    }
};

template<std::size_t Size>
void RingBuffer<Size>::put(const double &e) {
    ...
}

template<std::size_t Size>
void RingBuffer<Size>::get(double &e) {
    ...
}

template<std::size_t Size>
double RingBuffer<Size>::peek(std::size_t offset = 0) const {
    ...
}
```

RingBuffer<10> b;

RingBuffer b;

```
template<typename T, std::size_t N>
class RingBuffer {
    T data[N+1];
protected:
    std::size_t iput;
    std::size_t iget;
    static std::size_t wrap(std::size_t idx) {
        return idx % (N+1);
    }
public:
    RingBuffer()
        : iput(0), iget(0)
    {}
    bool empty() const {
        return (iput == iget);
    }
    bool full() const {
        return (wrap(iput+1) == iget);
    }
    std::size_t size() const {
        return (iput >= iget)
            ? iput - iget
            : iput + (N+1) - iget;
    }
    void put(const T &);
    void get(T &);
    T peek(std::size_t) const;
};
```

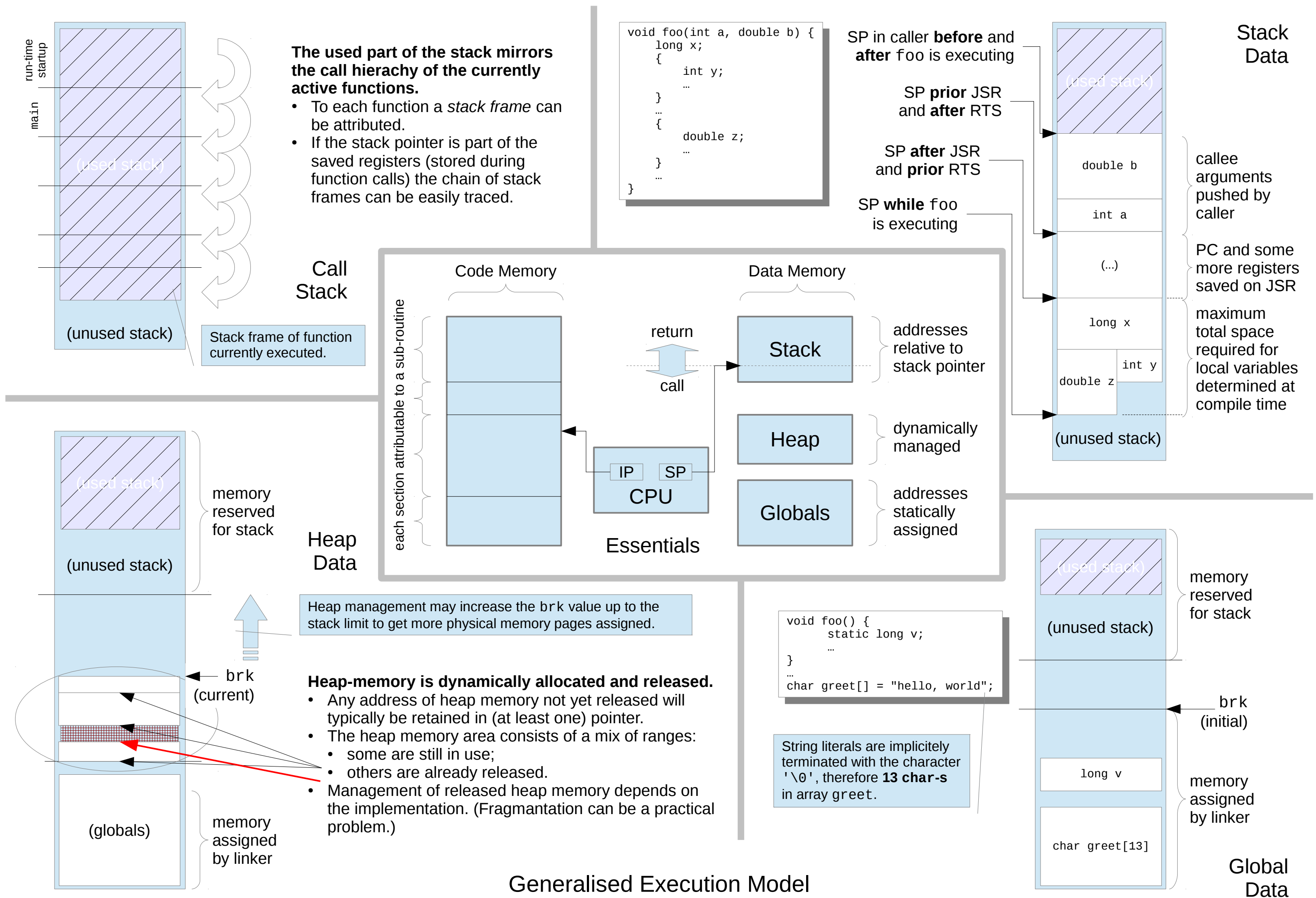
```
template<typename T, std::size_t N>
void RingBuffer<T, N>::put(const T &e) {
    if (full())
        iget = wrap(iget+1);
    assert(!full());
    data[iput] = e;
    iput = wrap(iput+1);
}
```

```
template<typename T, std::size_t N>
void RingBuffer<T, N>::get(T &e) {
    assert(!empty());
    e = data[iget];
    iget = wrap(iget+1);
}
```

```
template<typename T, std::size_t N>
T RingBuffer<T, N>::peek(std::size_t offset = 0) const {
    assert(offset < size());
    return data[wrap(idx + offset)];
}
```

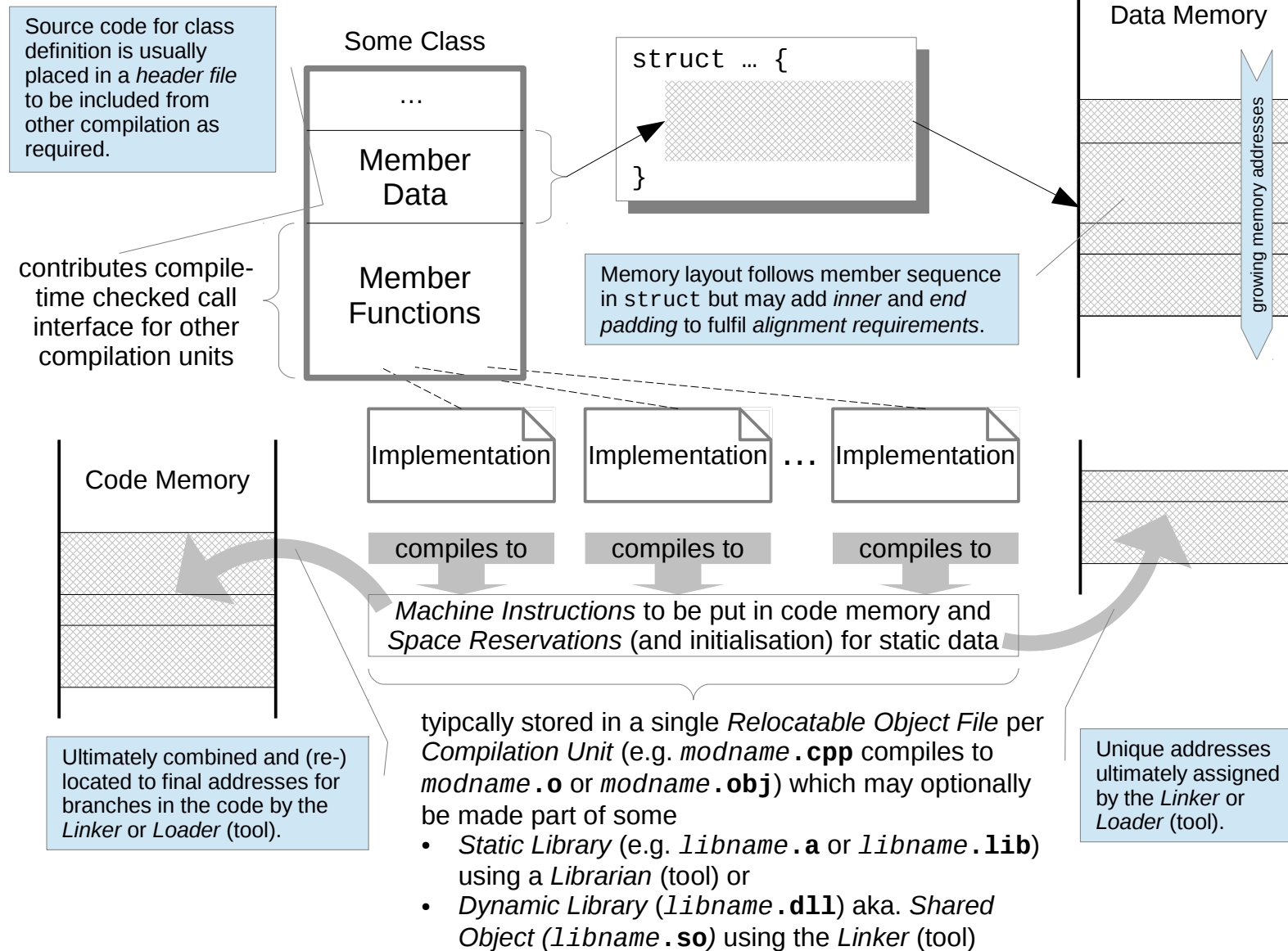
```
RingBuffer<double, 10> b;
RingBuffer<int, 10000> x;
RingBuffer<string, 42> y;
RingBuffer<MyClass, 9> z;
```

Parametrizing Types and Sizes

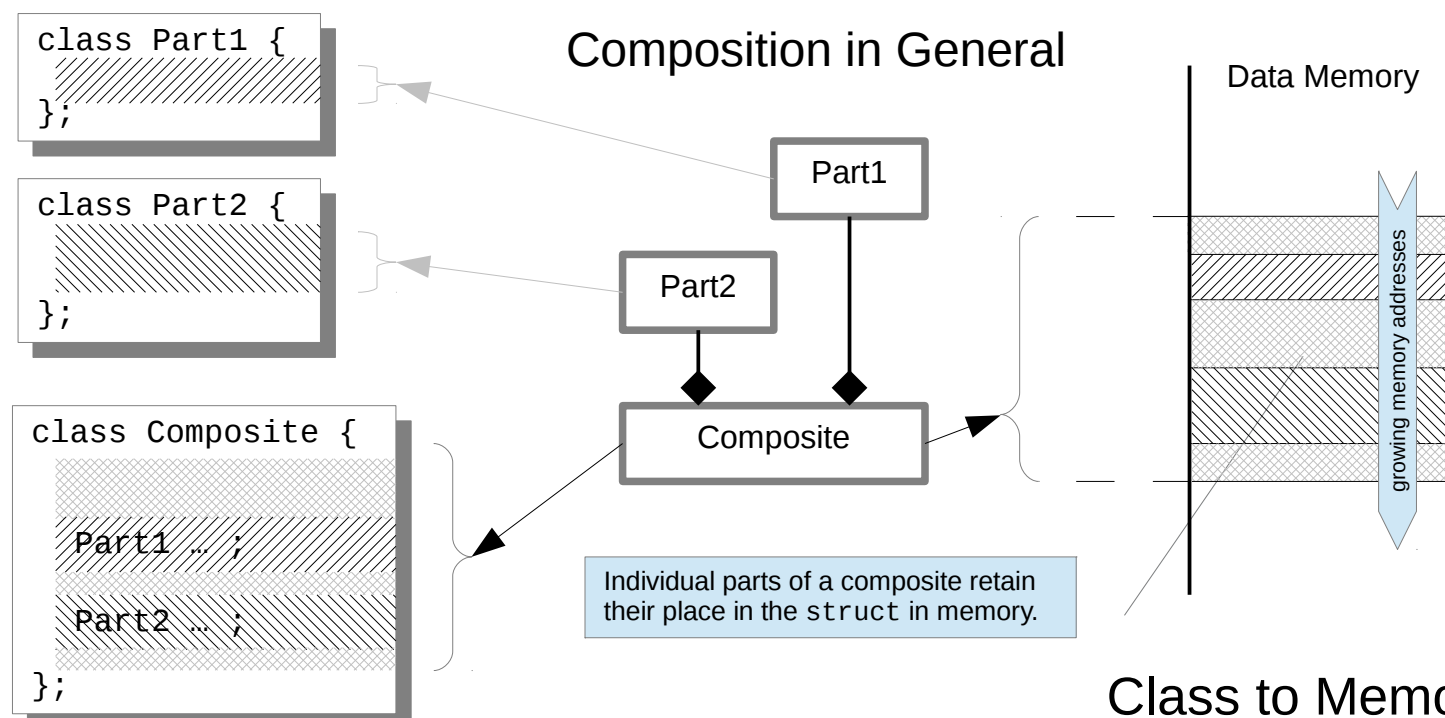


Generalised Execution Model

Mapping Classes to Code And Data



Composition in General



Class to Memory Mapping

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

The LSP is part of the type conversion rules built-in to the compiler (and a “no-op” at run-time for single inheritance): Any *Derived* can trivially be used as a *Base* because the memory layout of the latter is part of the former.

```

class Base {
};
    
```

```

class Derived
: public Base {
};
    
```

Base *

Derived *

LSP

Base

Derived

Public Base Class (Inheritance)

Data Memory

... versus ...

In absence of an explicit type conversion the compiler declines to accept a *Derived* where a *Base* is expected (despite the former contains the memory layout of the latter).

```

class Base {
};
    
```

```

class Derived
: private Base {
};
    
```

Base *

Derived *

no LSP

Base

Derived

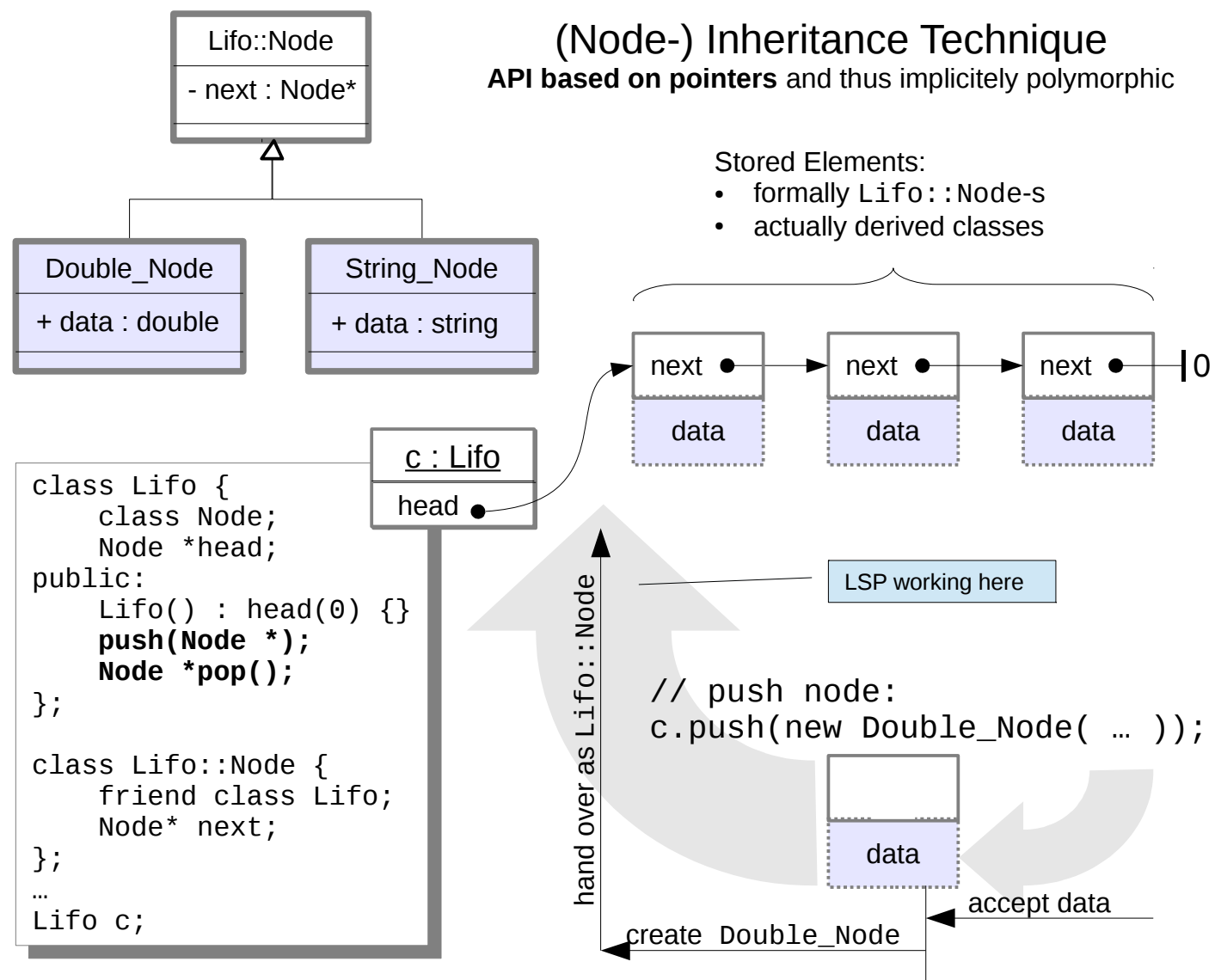
Privat Base Class (Composition)

Data Memory

The LSP – short for “Liskov Substitution Principle” - was formulated by *Barbara Liskov* and demands:

- Any object of a derived class should be a valid substitute for an object of its – direct or indirect – base classes.
- As long as only single inheritance is used the LSP is effectively a “no-op” in C++ since base class objects start at the same memory address as their derived classes.

As for private base classes there is no LSP in C++ they must be viewn as Composition, not Inheritance!



```

// pop node (double expected):
if (Double_Node *p = dynamic_cast<Double_Node *>(c.pop())) {
    // process node data:
    ... p->data ...
    // owning Node now!
    delete p;
}
  
```

take data

unexpected node types may cause memory leak with this coding style!

dynamic down-cast may return nullptr

extend to Double_Node*

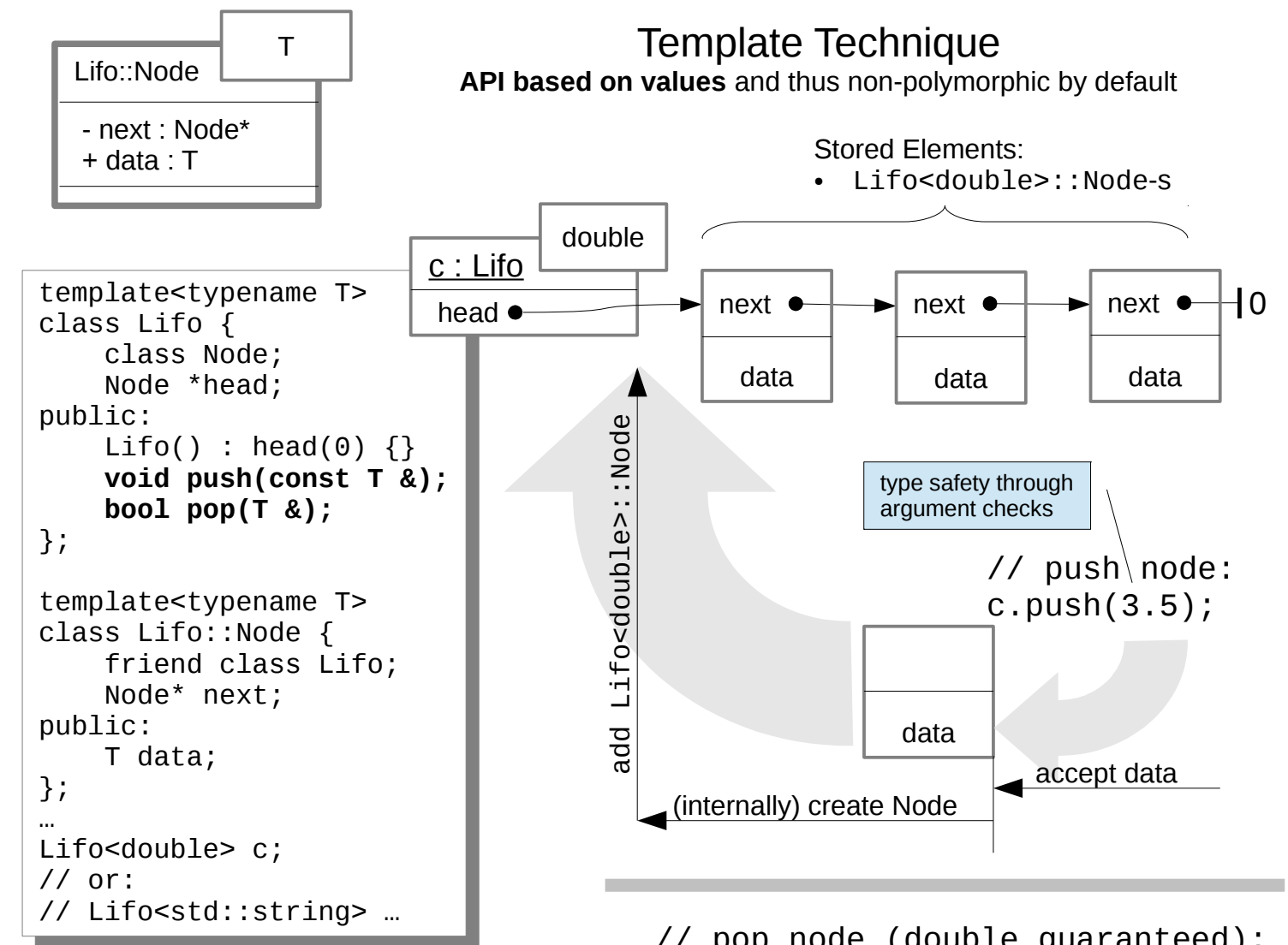
get Lifo::Node*

```

// zero run-time overhead (may cause undefined behavior):
... static_cast<Double_Node *>(p)->data ...
  
```

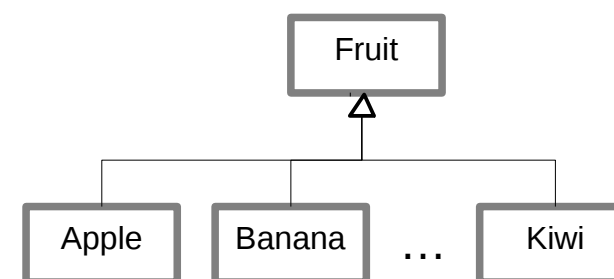
```

// safe short-hand (may throw):
... dynamic_cast<Double_Node &>(*p).data ...
  
```



```

// pop node (double guaranteed):
double node_data;
c.pop(node_data);
// process node data:
... node_data ...
  
```



Polymorphic Elements

```

Fifo<Fruit*> basket;
...
basket.push(new Apple( ... ));
...
fruit *f;
basket.pop(f);
  
```

for polymorphic elements containers must use pointers explicitly

now points to an Apple

Non-Polymorphic Elements

```

Apple a;
Banana b;
Fifo<Fruit> basket;
...
basket.push(a);
basket.push(b);
...
Fruit f;
basket.pop(f);
...
Basket.pop(a);
  
```

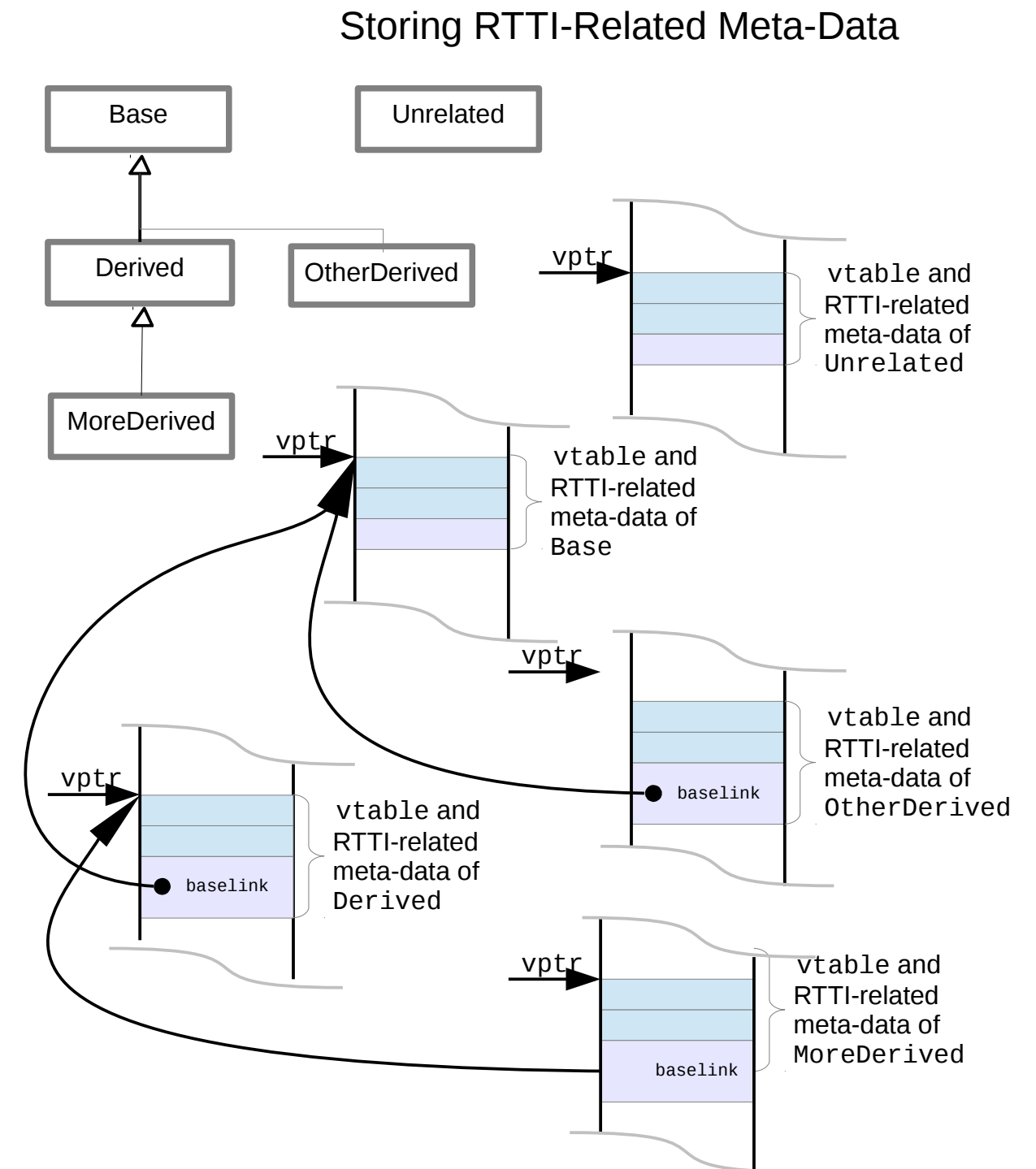
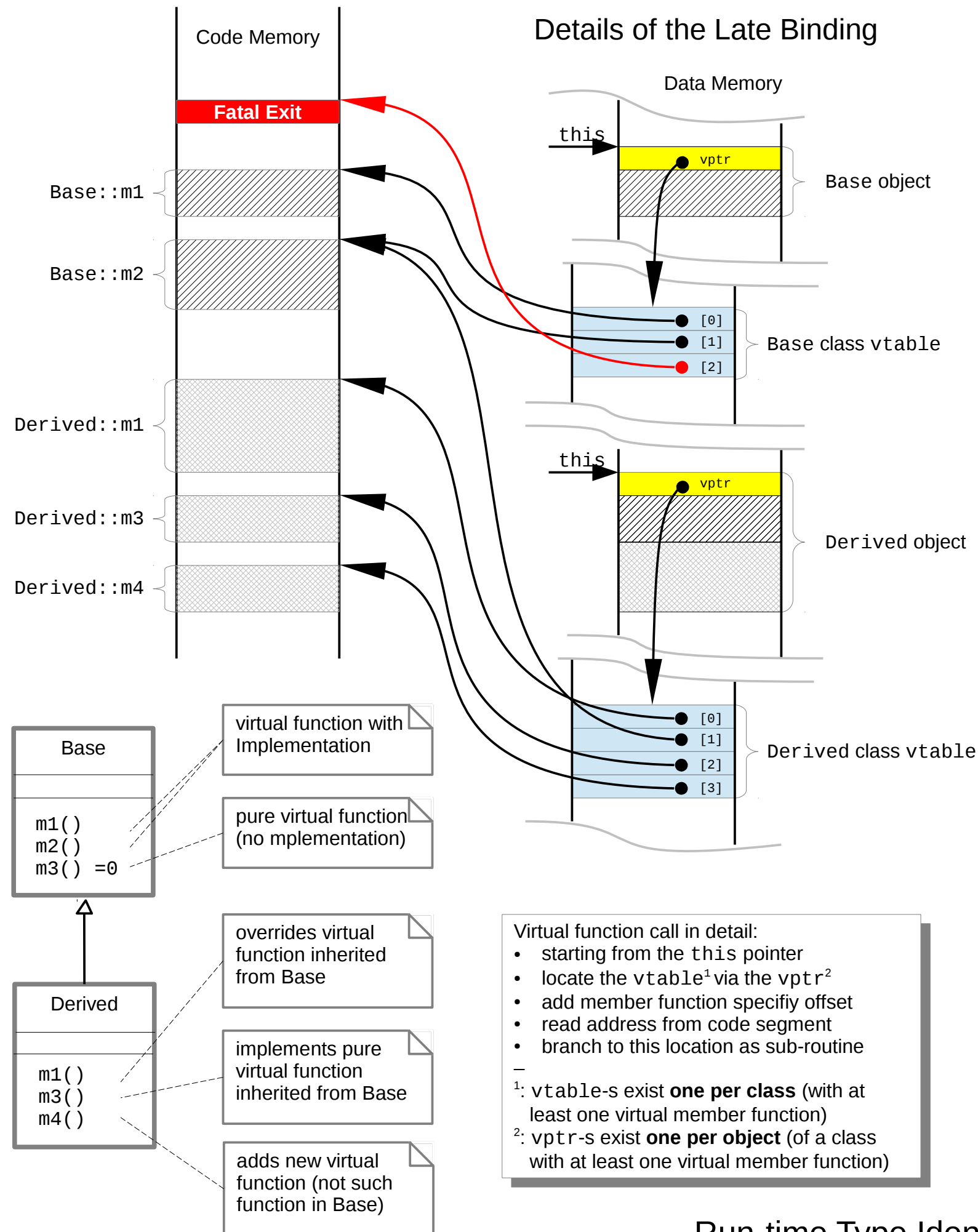
slices Apple-s and Banana-s to their Fruit base!

OK but, just a Fruit ... (Sorry Sir, no Banana flavour today!)

does not compile because all what is returned are Fruit-s

Container Implementation Techniques

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>



RTTI is limited classes with at least one virtual member function:

- This avoids overhead which would otherwise occur on per object.
- Meta-data is stored in the vicinity of the vtable.

RTTI-related meta-data is used by:

- `dynamic_cast` – checks for given class or derived (“usable as”):
 - in pointer syntax `nullptr` is returned in case of failure;
 - in reference syntax an exception is thrown in case of failure.
- `typeid` – checks for exact class and gives some more information (see `struct std::type_info` defined in header `<typeinfo>` for details).

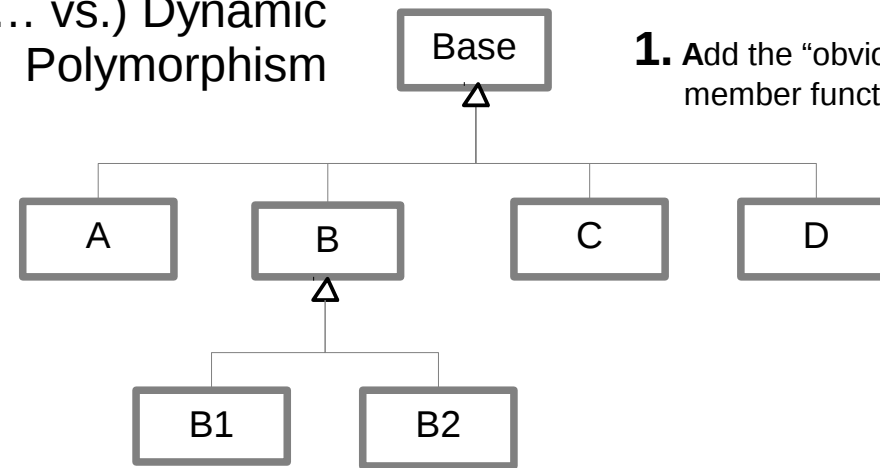
Run-time Type Identification

Type-dependent Flow of Control

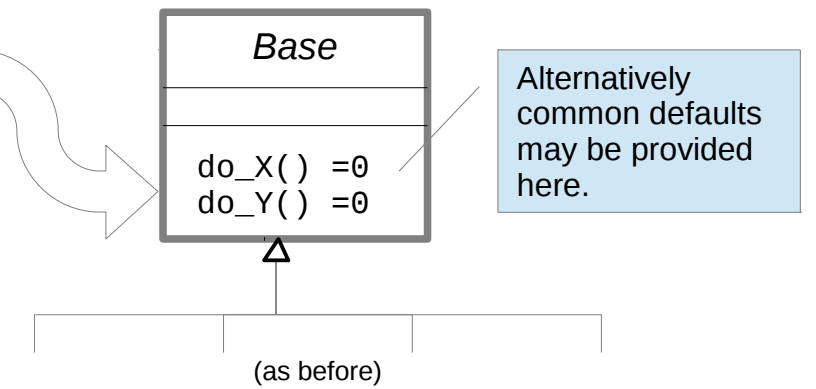
```
void foo(Base &r) {
    ...
    if (auto p = dynamic_cast<A*>(&r)) {
        ...
    }
    if (auto p = dynamic_cast<B1*>(&r)) {
        ...
    }
    if (auto p = dynamic_cast<B2*>(&r)) {
        ...
    }
    if (auto p = dynamic_cast<C*>(&r)) {
        ...
    }
    if (auto p = dynamic_cast<D*>(&r)) {
        ...
    }
    ...
    // combining B1 and B2
    if (auto p = dynamic_cast<B*>(&r)) {
        ...
    }
    ...
}
```

```
void foo(Base &r) {
    ...
    if (typeid(r) == typeid(A)) {
        ...
    }
    if (typeid(r) == typeid(B1)) {
        ...
    }
    if (typeid(r) == typeid(B2)) {
        ...
    }
    if (typeid(r) == typeid(C)) {
        ...
    }
    if (typeid(r) == typeid(D)) {
        ...
    }
    ...
    // combining B1 and B2
    if (typeid(r) == typeid(B1)
        || typeid(r) == typeid(B2)) {
        ...
    }
    ...
}
```

(... vs.) Dynamic Polymorphism



1. Add the “obviously missing” member functions to Base:



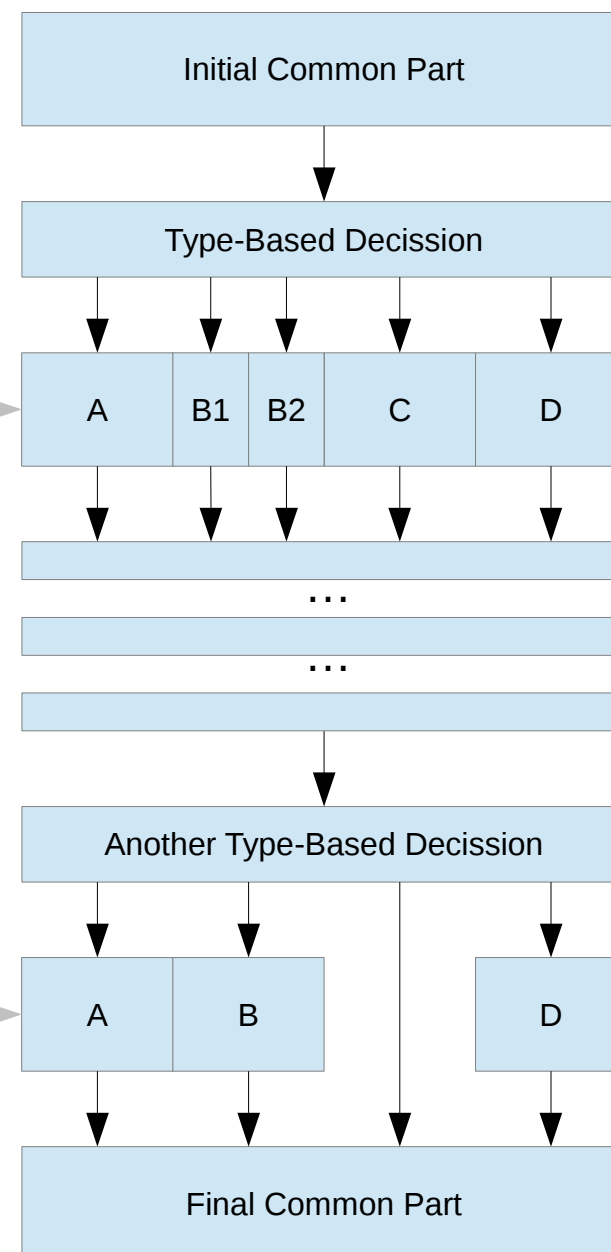
2. Move actions from multiway branches to member function implementations:

```
void A::do_X() {
    ...
}
void B1::do_X() {
    ...
}
void B2::do_X() {
    ...
}
void C::do_X() {
    ...
}
void D::do_X() {
    ...
}
```

```
void A::do_Y() {
    ...
}
void B::do_Y() {
    ...
}
void C::do_Y() {
    /*empty*/
}
void D::do_Y() {
    ...
}
```

Data can be shared easily in privacy.

The need for sharing data may weaken information hiding.



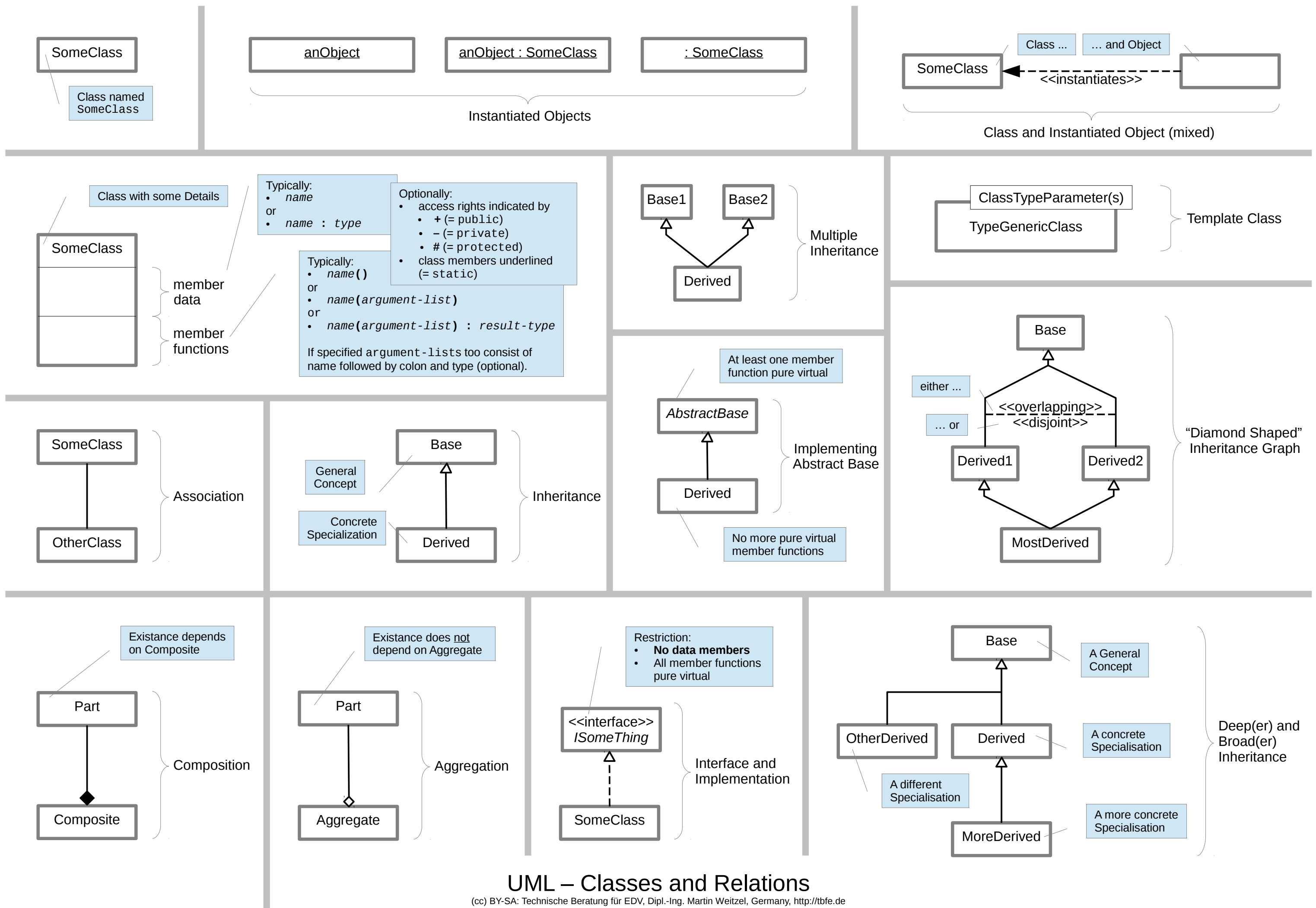
do X

do Y

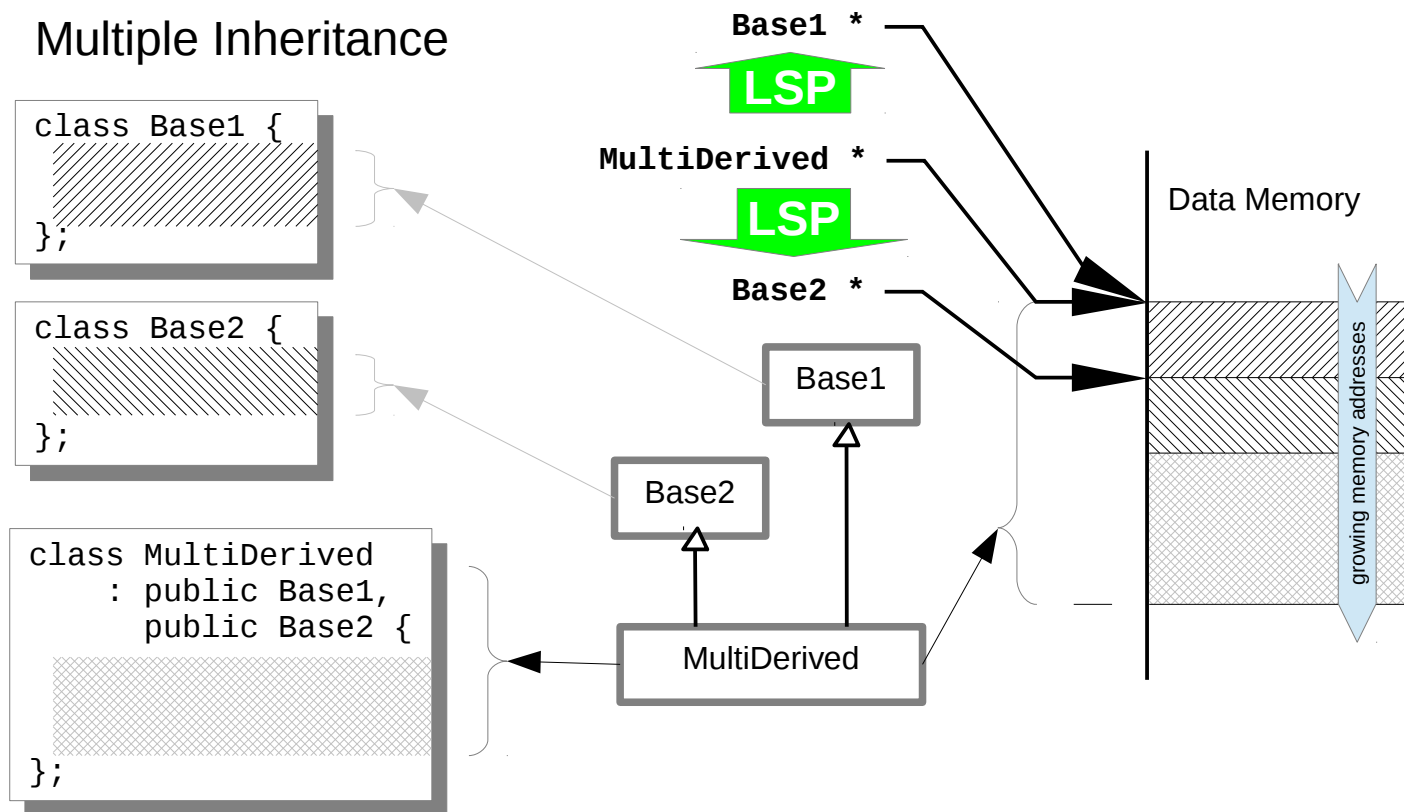
3. Replace multiway branches with member function calls:

```
...
r.do_X();
...
r.do_Y();
...
```

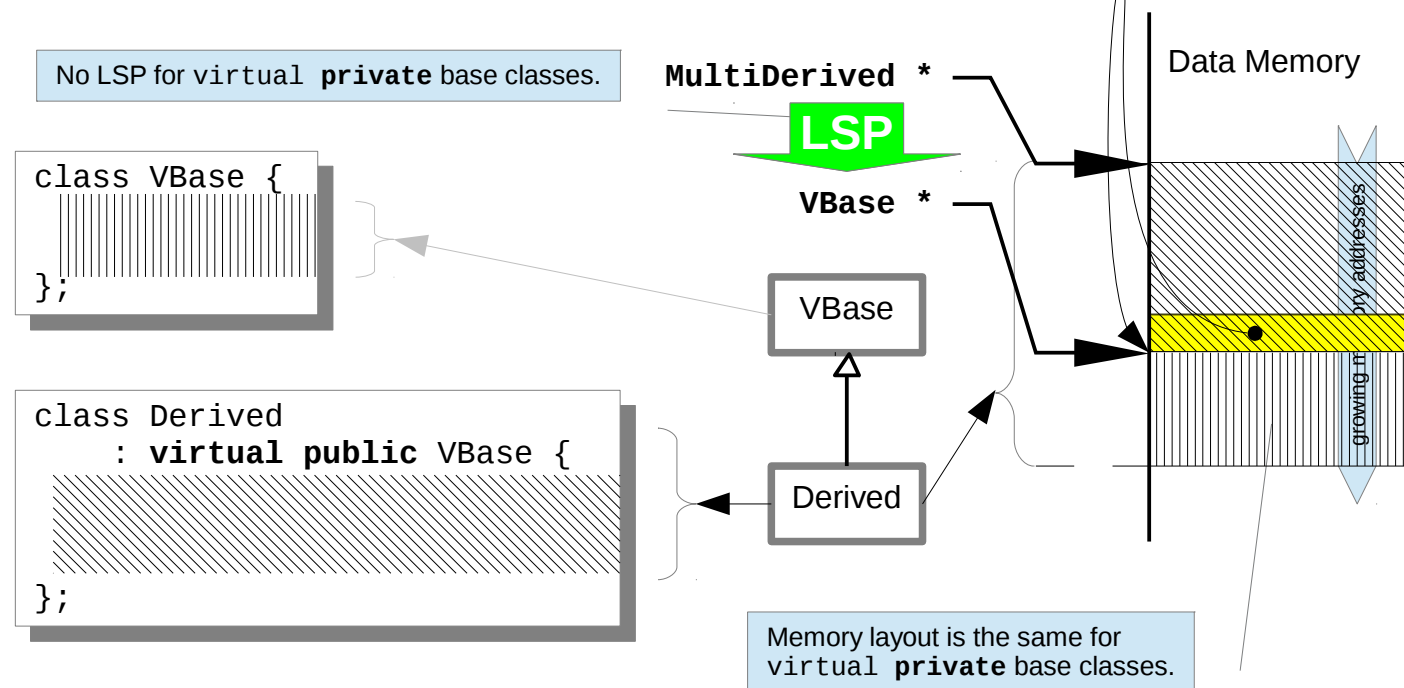
Type-Based Multiway Branching



Multiple Inheritance



Virtual Base Class

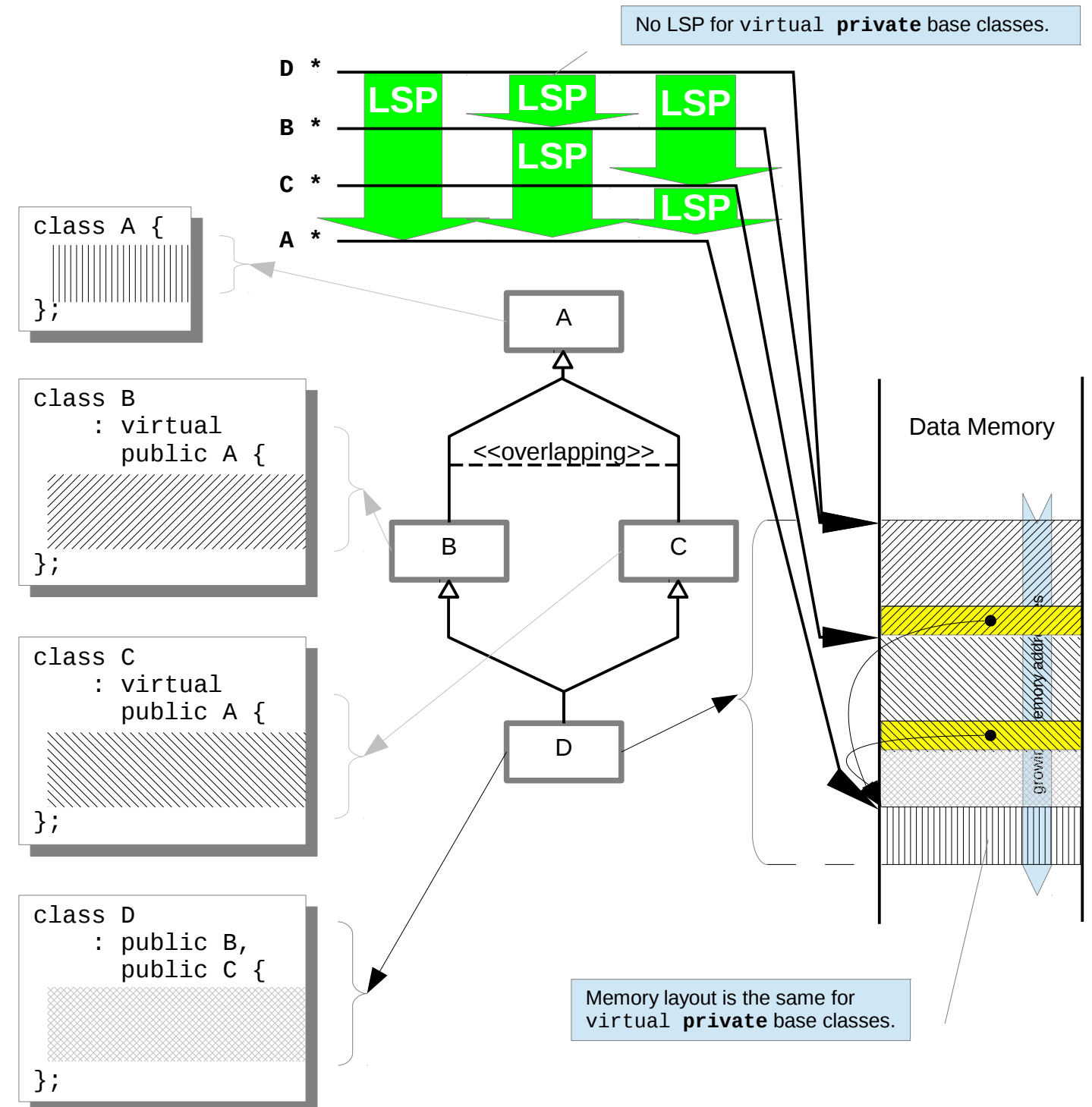


A virtual base class introduces additional overhead in the derived class:

- space is allocated for an pointer which points to the base class part;
- all access to the base class part is indirect using this pointer.

As far as is shown virtual base classes have no advantage.

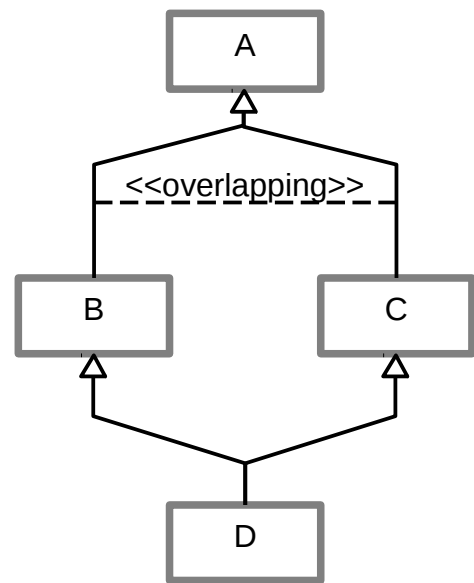
Overlapping Common Base Class



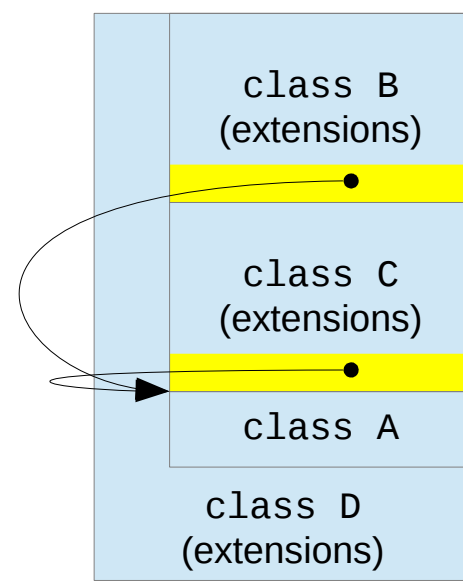
Virtual base classes are the mechanism to make a common base in a “diamond-shaped” inheritance relationship overlapping (see A above).

- This has to be prepared by the classes at the intermediate level (B and C above).
- The most derived class (D above) does not use virtual bases – it finds its direct bases at fixed offsets.
- These bases refer to their base via the embedded pointer (see left side).
- Both pointers are set to point to the same (embedded) base object.

Multiple Inheritance and Virtual Base Classes



UML Class Graph

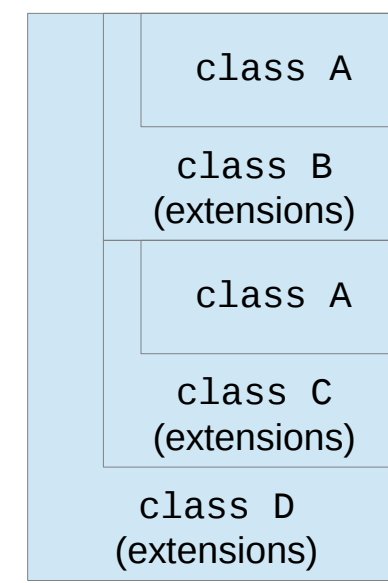


Member Data to Memory Mapping

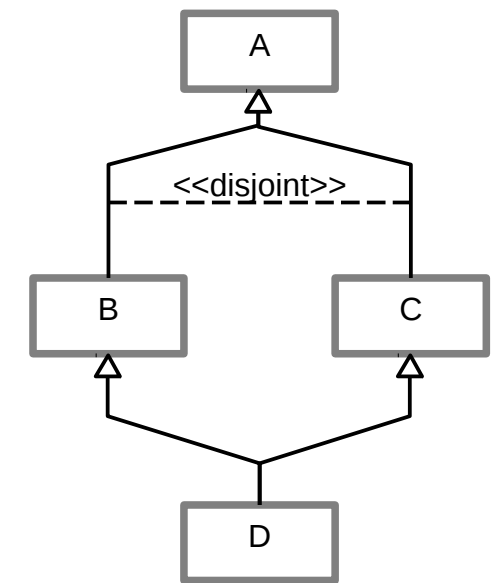
Up-Casts by LSP

Automatic Type Conversions

<i>to</i> ←	<i>from</i>	→ <i>to</i>
A	A	A
A	B	A
A	C	A
A, B, C	D	B, C



Member Data to Memory Mapping



UML Class Graph

```
class A {
    ...
};
class B : virtual public A {
    ...
};
class C : virtual public A {
    ...
};
class D : public B, public C {
    ...
};
```

C++ Source

Creation and Destruction of D objects

C++ Source

```
class A {
    ...
};
class B : public A {
    ...
};
class C : public A {
    ...
};
class D : public B, public C {
    ...
};
```

```
A::A( ... ) { ... };
```

Virtual base
constructed from most derived class
(trying default construction if no explicit constructor)

```
B::B( ... )
: A( ... )
{ ... };
```

```
C::C( ... )
: A( ... )
{ ... };
```

```
D::D( ... )
: A( ... ), B( ... ), C( ... )
{ ... };
```

Order of Constructor Calls

A::A(...)	MI-List, then Body
B::B(...)	(remaining) MI-List except A::A(...), then Body
C::C(...)	(remaining) MI-List except A::A(...), then Body
D::D(...)	MI-list, then Body

Order of Destructor Calls

D::~~D()	Body, chaining to
C::~~C()	Body, chaining to
B::~~B()	Body, chaining to
A::~~A()	

Special rule for calling virtual base class constructors:

- executed when a B or C object is created stand-alone;
- ignored when a B or C base of class of D is created.

Order of Constructor Calls

A::A(...)	base of B	MI-List, then Body
B::B(...)		(remaining) MI-List, then Body
A::A(...)	base of C	MI-List, then Body
C::C(...)		(remaining) MI-List, then Body
D::D(...)		(remaining) MI-List, then Body

Order of Destructor Calls

D::~~D()		Body, chaining to
C::~~C()		Body, chaining to
A::~~A()	base of C	Body, chaining to
B::~~B()		Body, chaining to
A::~~A()	base of B	Body

No special rule for calling (non-virtual) base class constructors:

- each class cares for its direct base(s);
- **no knowledge wrt. indirect bases.**

```
A::A( ... ) { ... };
```

```
A::A( ... ) { ... };
```

```
B::B( ... )
: A( ... )
{ ... };
```

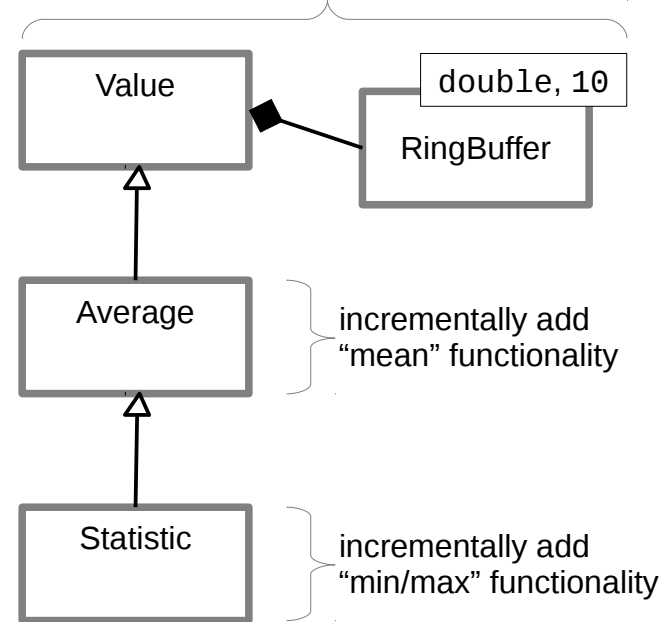
```
C::C( ... )
: A( ... )
{ ... };
```

```
D::D( ... )
: B( ... ), C( ... )
{ ... };
```

Diamond Shaped Inheritance

Reusing Adapted Component

building on existing flexible component

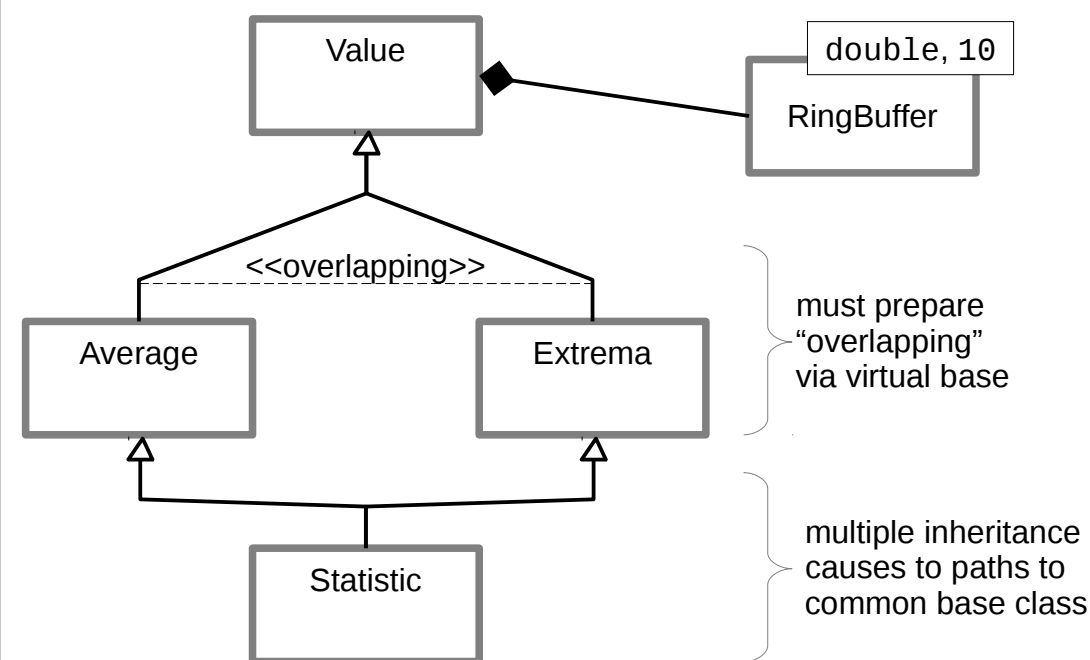


Simple design using:

- template (*Instantiation of RingBuffer*)
- composition (*Value has a RingBuffer*)
- base classes (*Average is a Value and Statistic is a Average*)

offers flexibility in combinations

Diamond-Shaped Inheritance

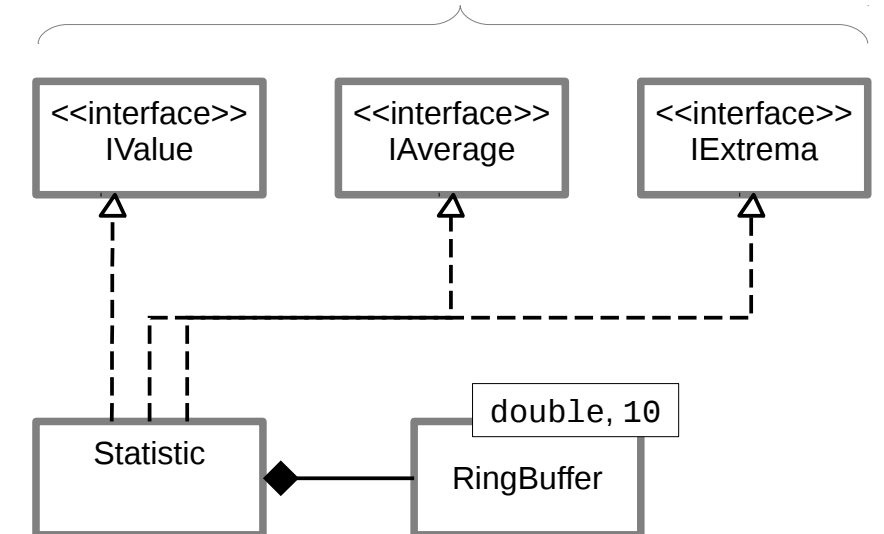


More flexible design with "diamond shaped" inheritance:

- each of the classes (*Value, Average, Extrema, Statistic*) may be used on its own
- intermediate classes (*Average, Value*) have to pay the "price" ...
- ... for simple re-use in the most derived class (*Statistic*)

Three Interfaces

simplifies view for specific sub-systems

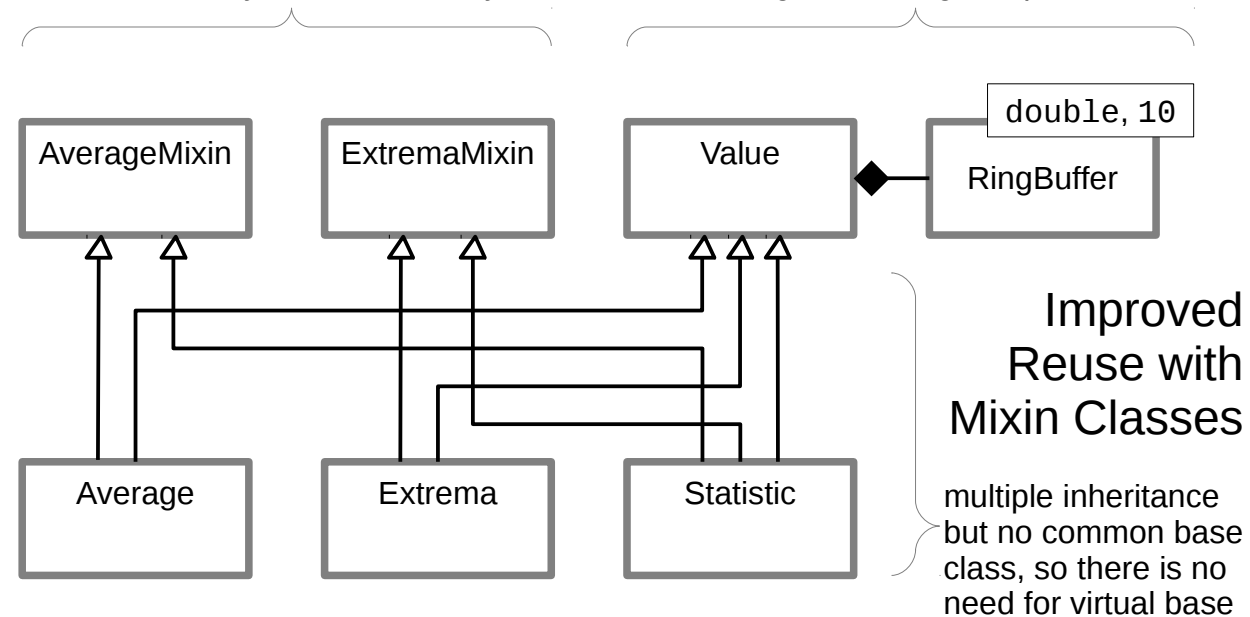


Alternative design with interfaces hides complexity from clients that do not need to know details:

- some clients may only need to handle Values (→ to know *IValue* is sufficient)
- others may only need to handle Averages (→ to know *IAverage* is sufficient)
- Yetl others may only need to handle Extrema (→ to know *IExtrema* is sufficient)

incrementally add functionality

building on existing components

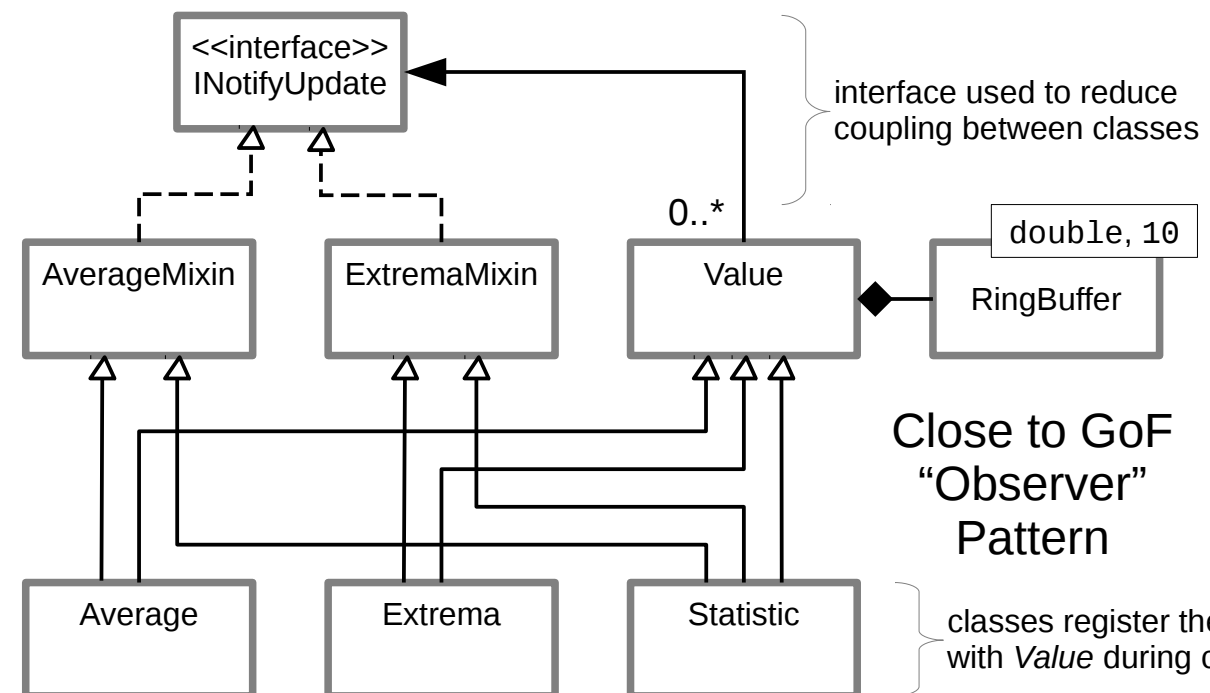


Improved Reuse with Mixin Classes

multiple inheritance but no common base class, so there is no need for virtual base

More elaborate design:

- flexibility achieved with "mixin" classes
- multiple inheritance but not "diamond shaped"

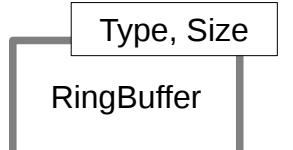


Close to GoF "Observer" Pattern

classes register themselves with *Value* during construction

Still more elaborate design:

- Mixins notified via generic interface
- *Value* only handles *INotifyUpdate*

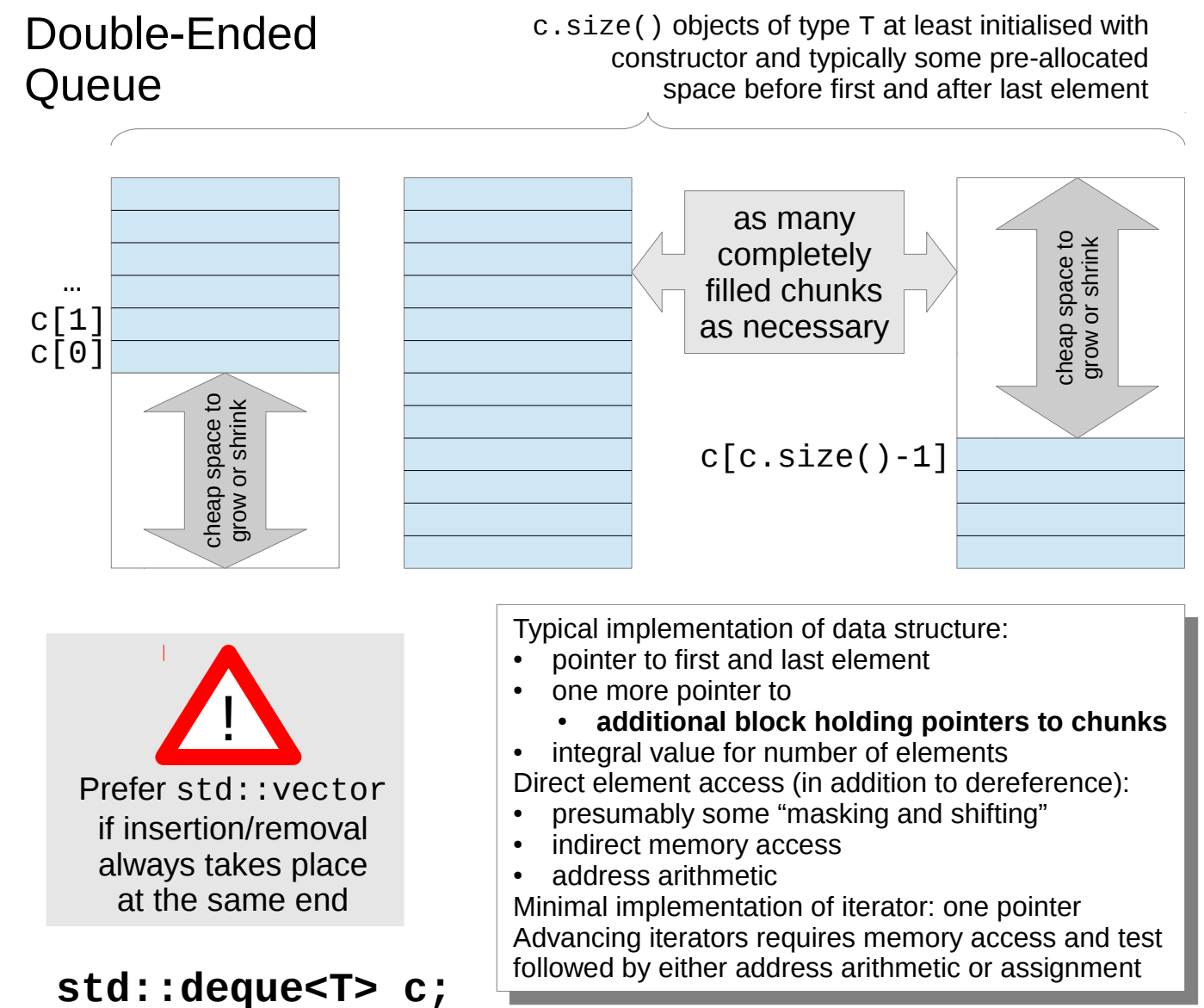
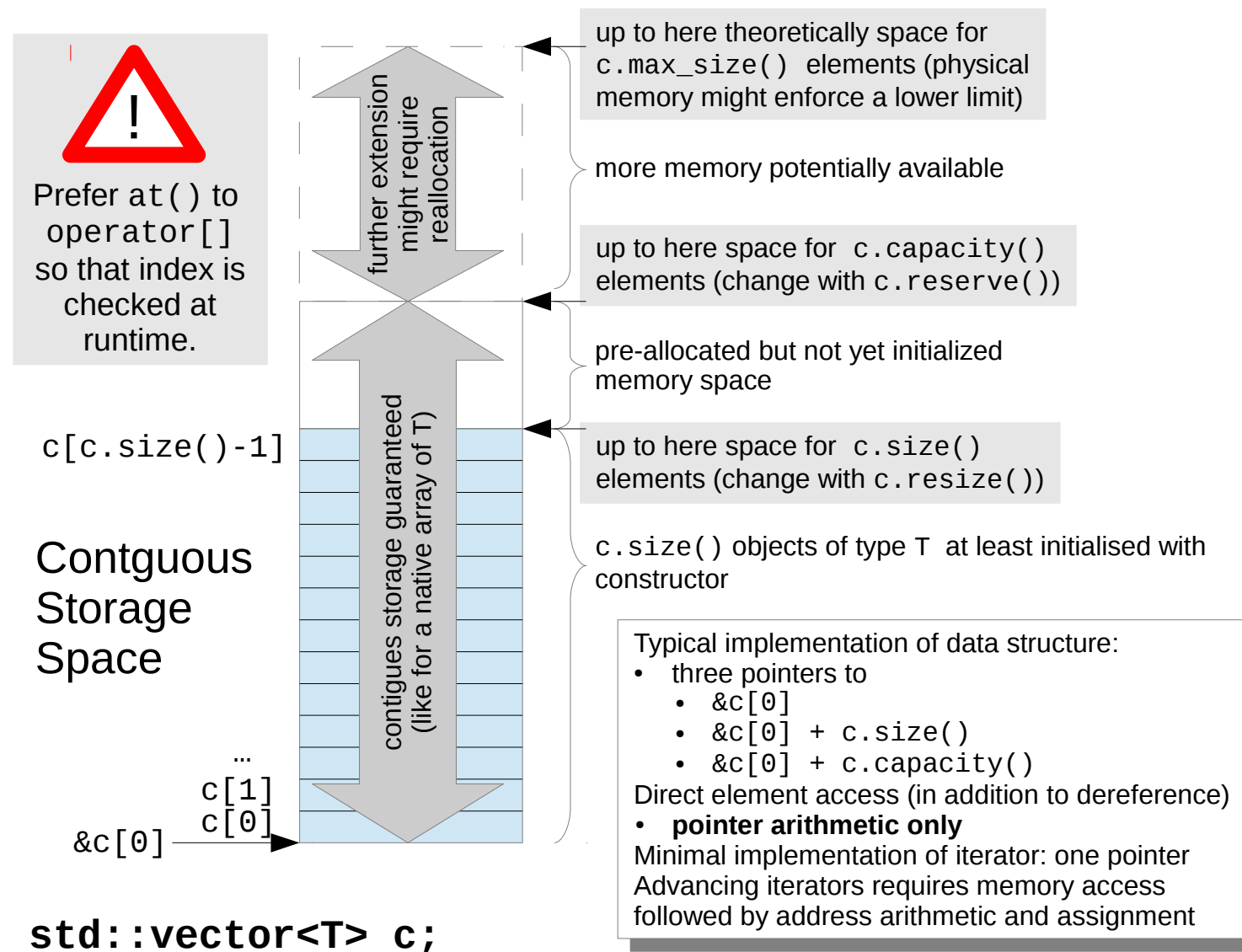


Reusable Component

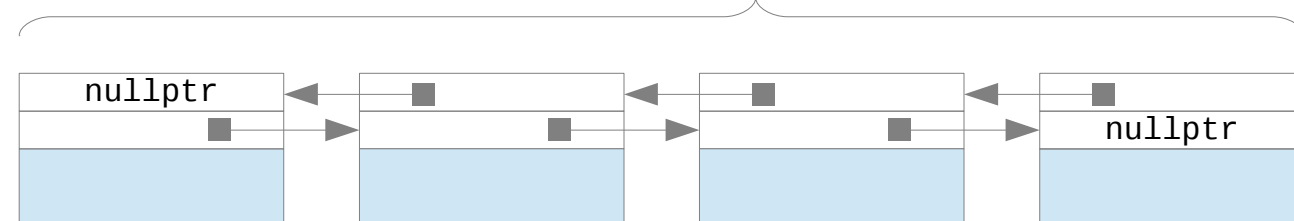
Generic class parametrized in

- **type** and
- **number** of elements

Examples – Classes and Relations



std::list<T> c; `c.size()` objects of type `T` at least initialised with constructor



Typical implementation:

- two pointers per element**
- pointer to first and last element
- integral value for number of elements

Direct element access not supported!

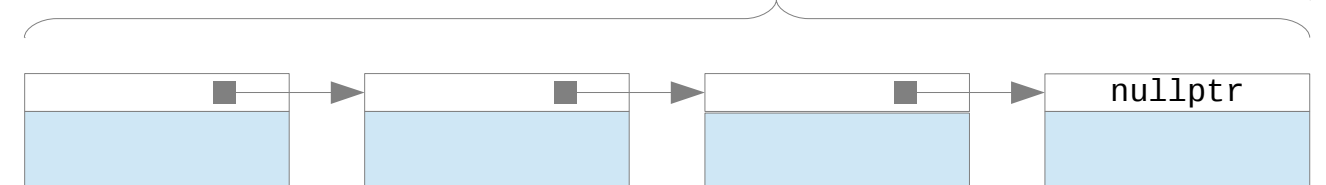
Minimal implementation of iterator: one pointer

Advancing iterators requires memory access followed by assignment

Double Linked List

Substantial overhead if `sizeof(T)` is small.

std::forward_list<T> c; objects of type `T` initialised with constructor



Singly Linked List

Use `c.empty()` to check whether elements exist.

Typical implementation:

- one pointer per element**
- only pointer to first element
- number of elements not stored!**

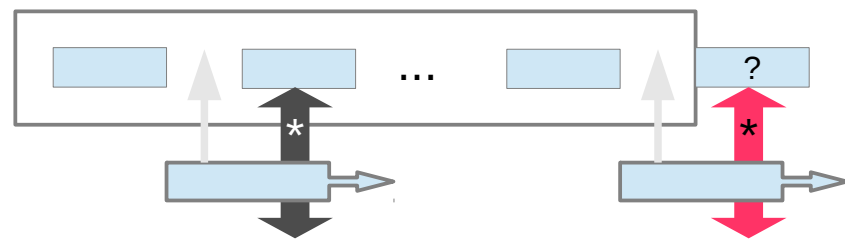
Direct element access not supported!

Minimal implementation of iterator: one pointer

Advancing iterators requires memory access followed by assignment

STL – Sequence Container Classes

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

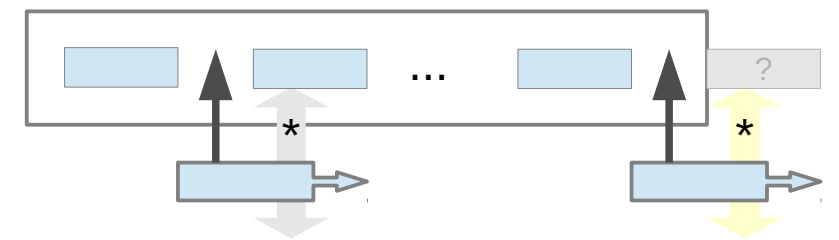


Emphasizing Element Access:

- Iterator points **onto** elements
- **must not be dereferenced in end position!**



Iterator
for Empty
Container



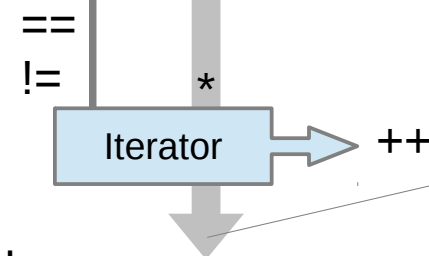
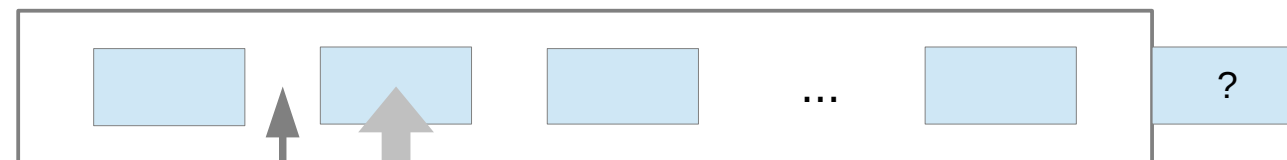
Emphasizing Current Position:

- Iterator points **between** elements
- **accessed element lies in direction of move**

Order defined by

- **insertion** (deletion, explicit sorting ...) for vector, deque, list, forward_list
- **element order** for set and multiset
- **key order** for map and multimap
- implementation for unordered_-containers(i.e. technically unspecified)

(front) container filled with some elements (back)

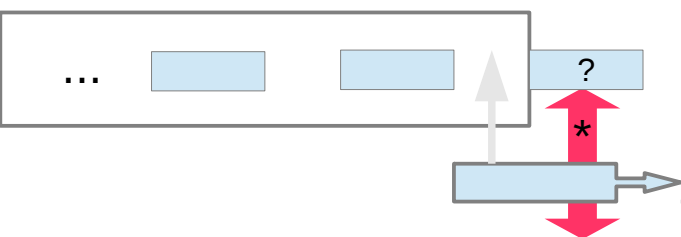
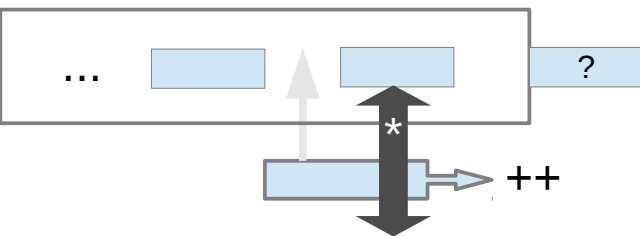


Increment operation moves iterator by one element

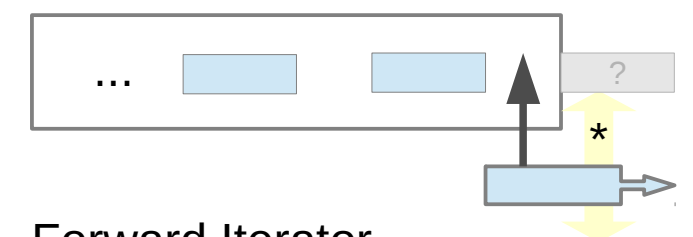
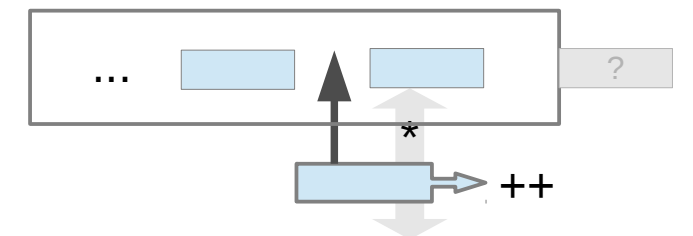
Iterator dereferencing accesses

- element value for most containers ...
- ... **except** for all kinds of map-s where a struct with elements first (key) and second (value) is returned

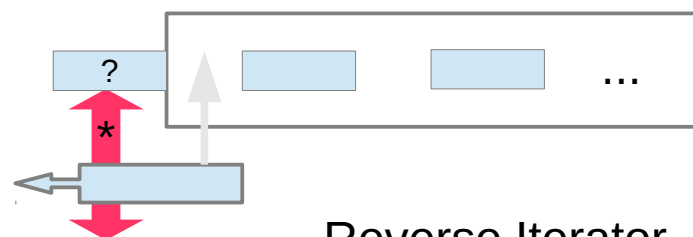
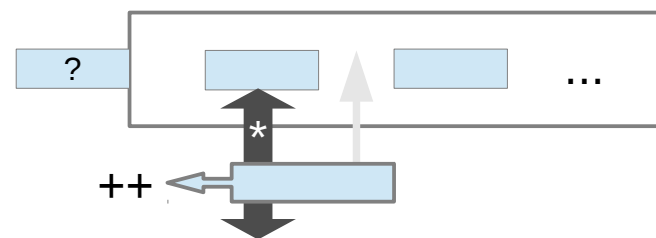
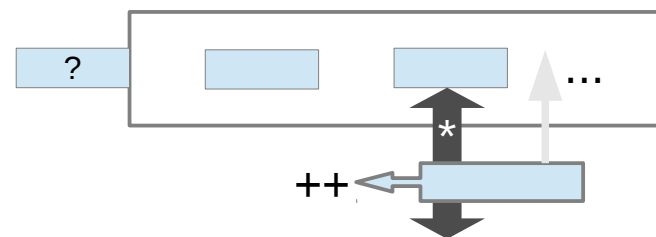
Essentials



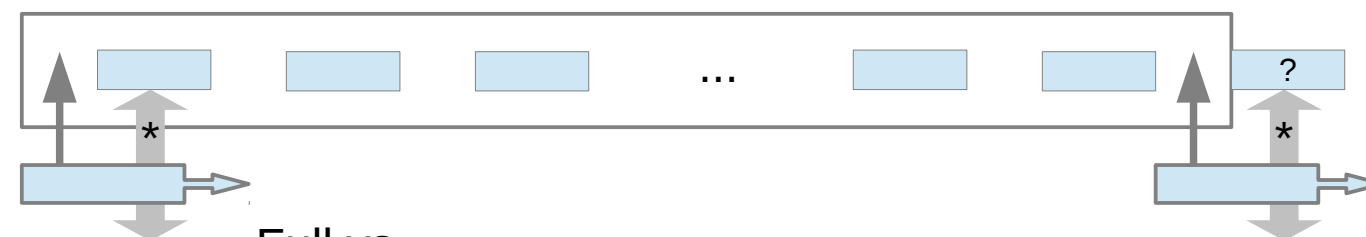
Forward Iterator



Forward Iterator

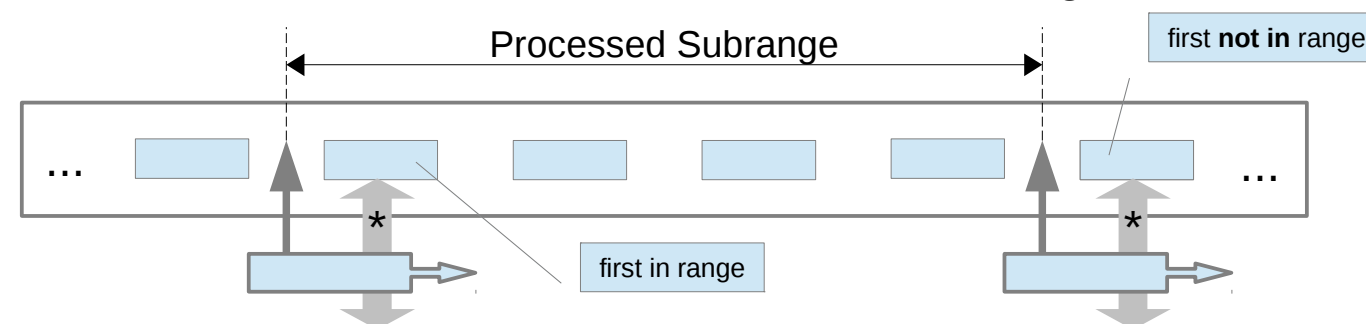


Reverse Iterator



Full vs. ...

... Partial Container Processing

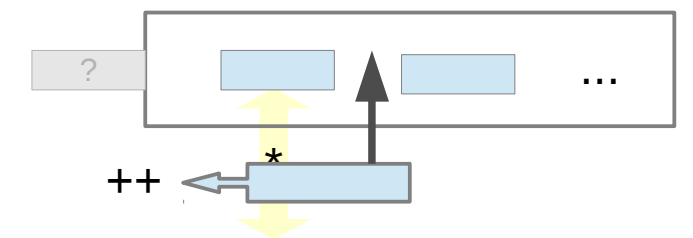
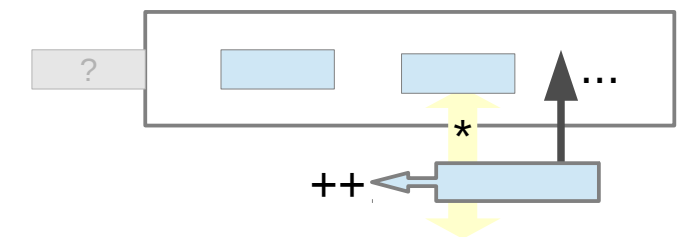


first in range

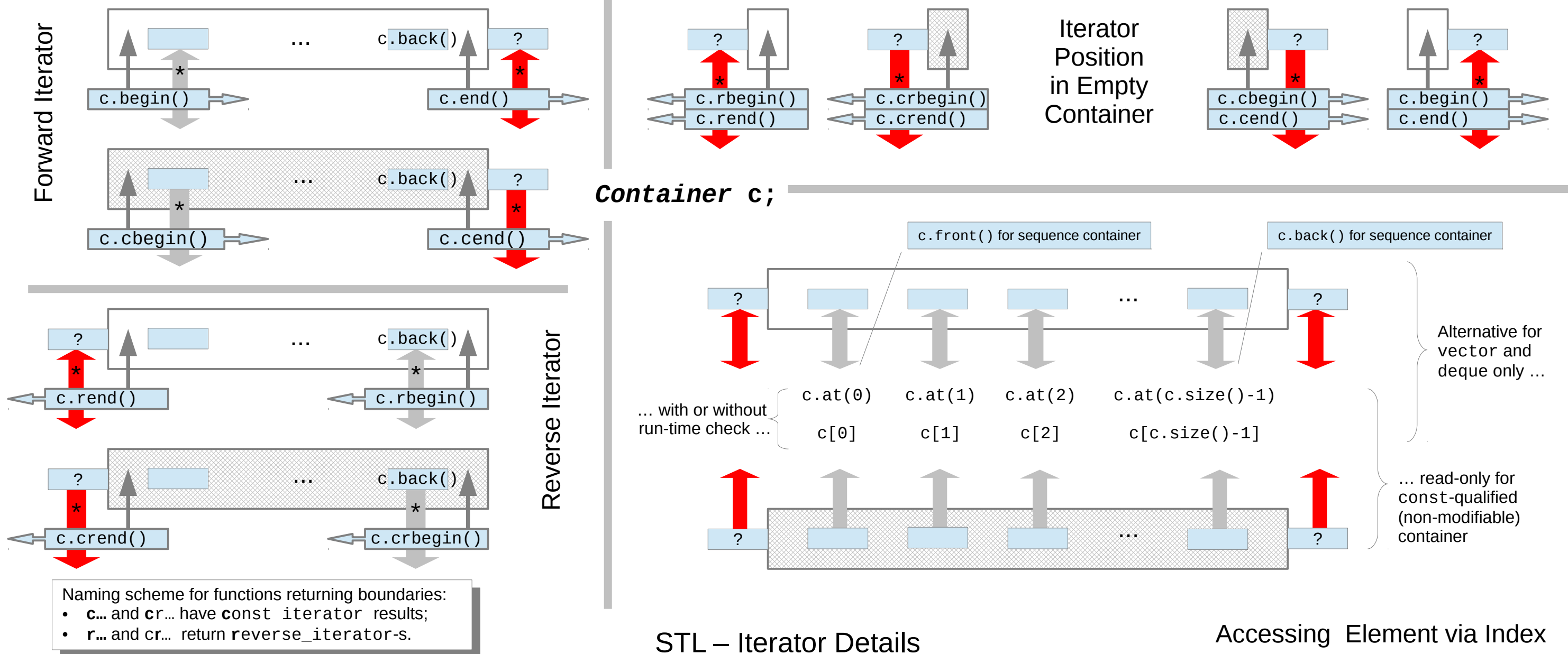
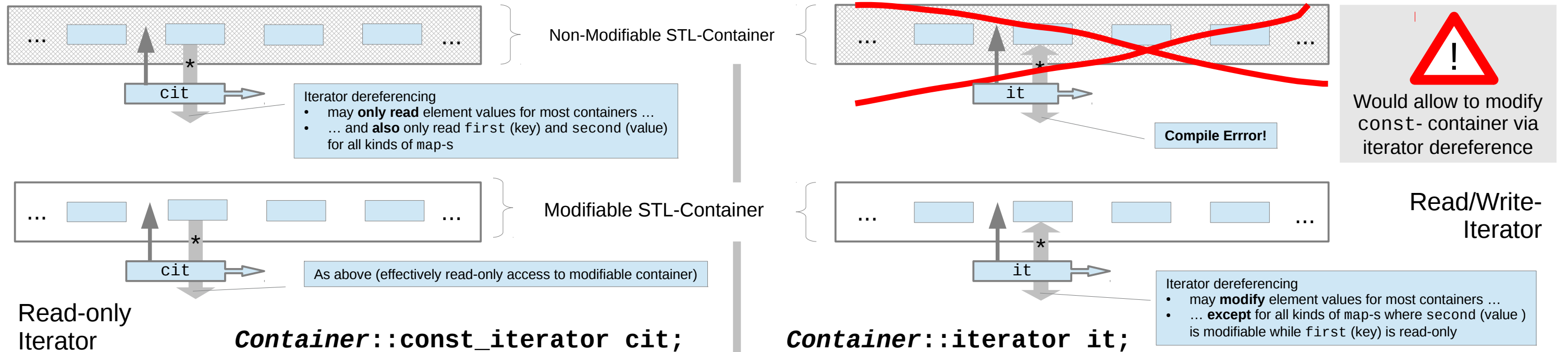
first not in range

STL – Container Iterators

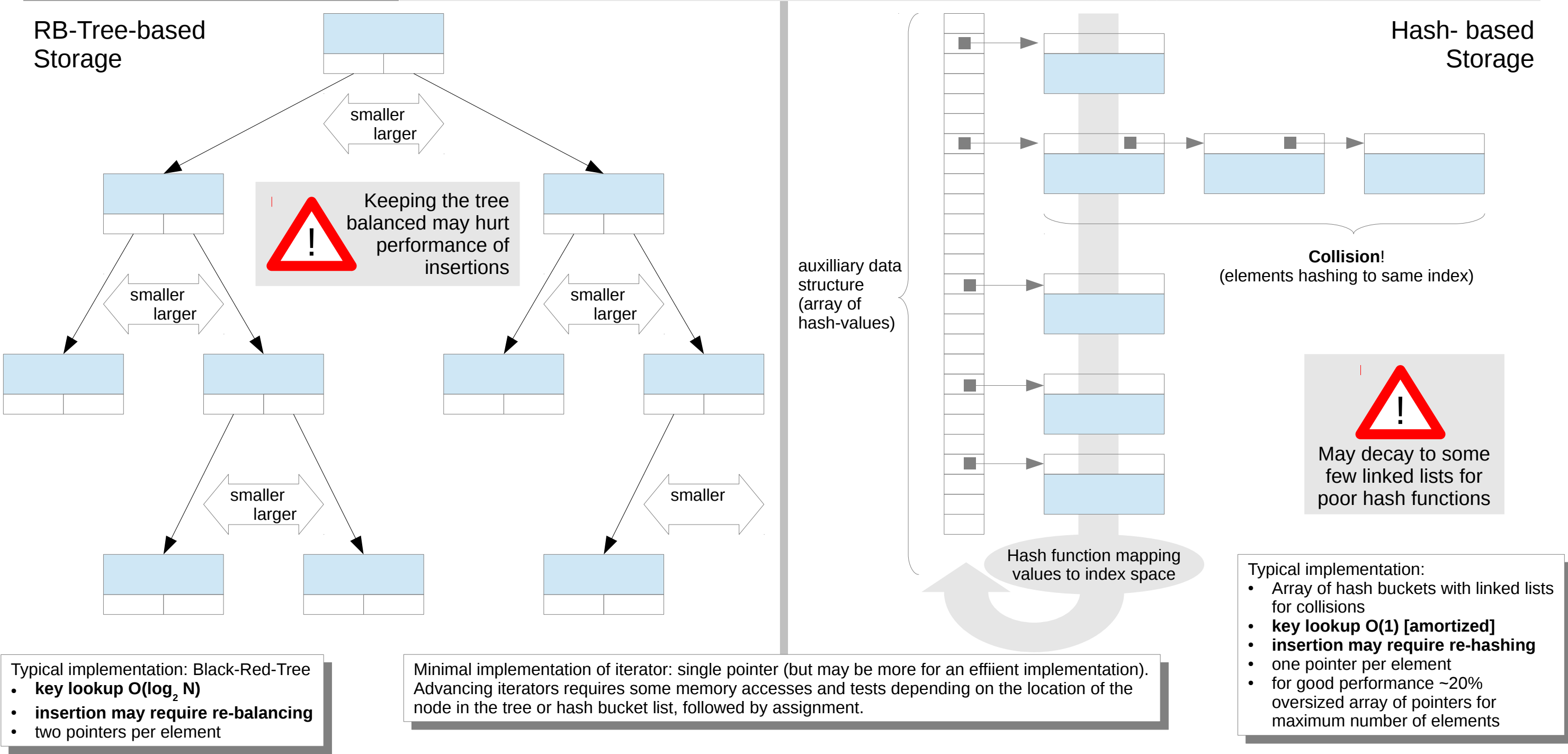
(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>



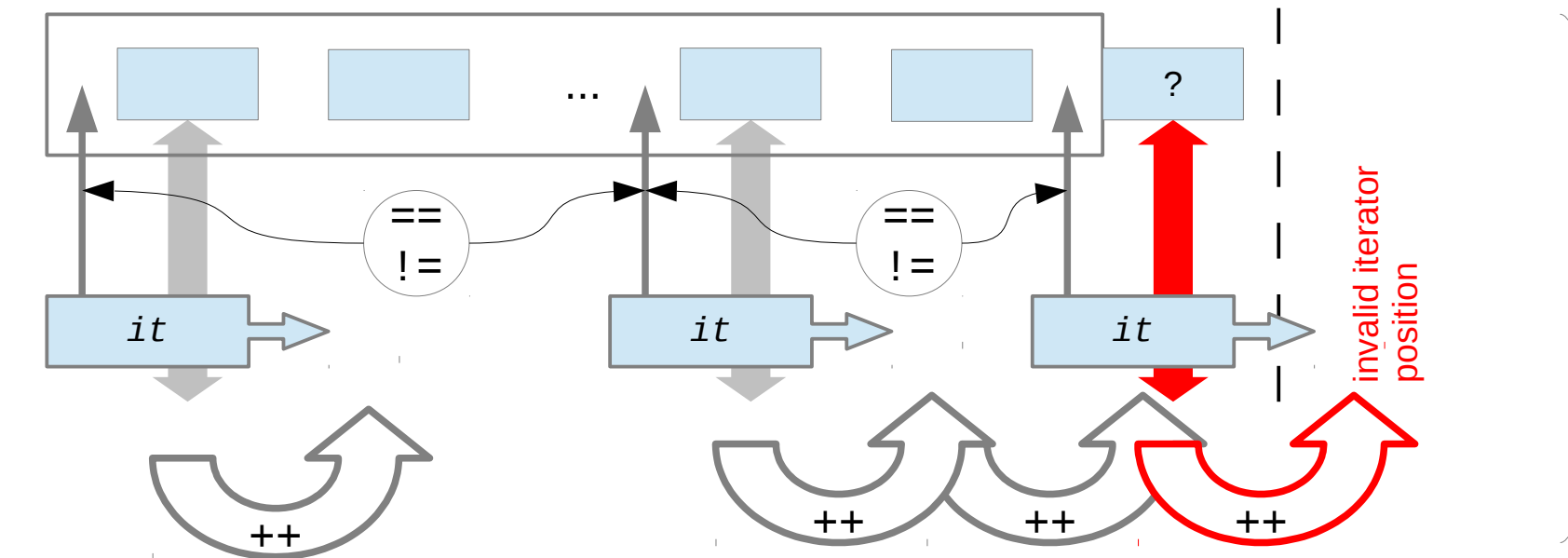
Reverse Iterator



Contained elements	STL Class Name		Restrictions
objects of type T	<code>std::set</code>	<code>std::unordered_set</code>	unique elements guaranteed
	<code>std::multiset</code>	<code>std::unordered_multiset</code>	multiple elements possible (comparing equal to each other)
pairs of objects of type T1 (key) and type T2 (associated value)	<code>std::map</code>	<code>std::unordered_map</code>	unique keys guaranteed
	<code>std::multimap</code>	<code>std::unordered_multimap</code>	multiple keys possible (comparing equal to each other)

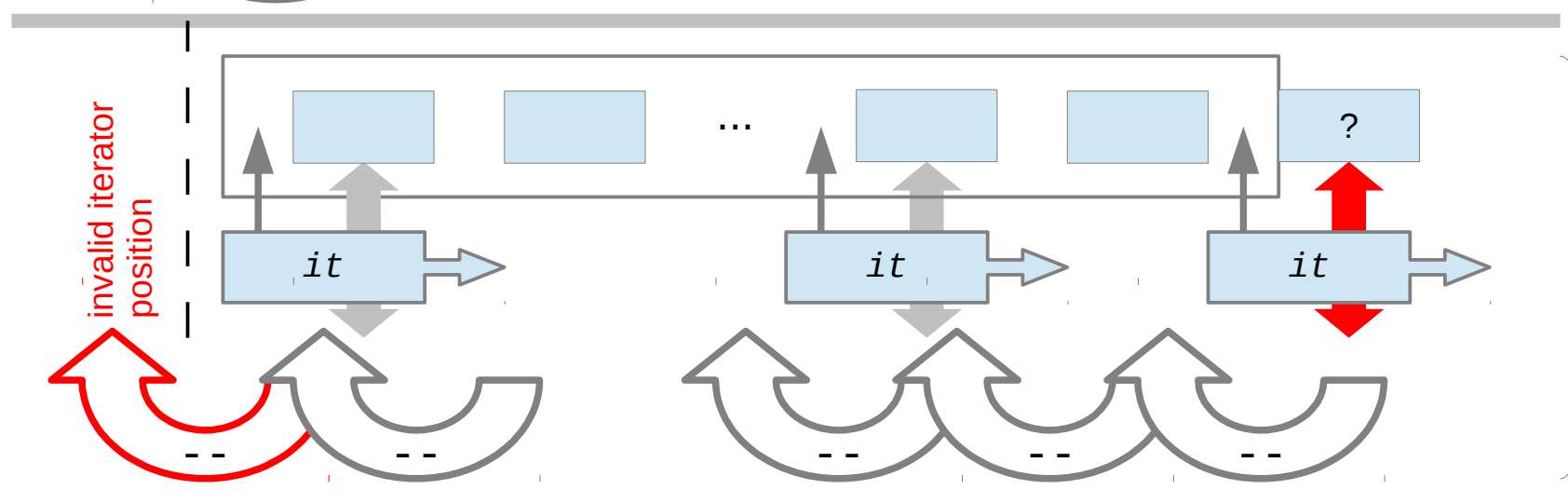


STL – Associative Container Classes



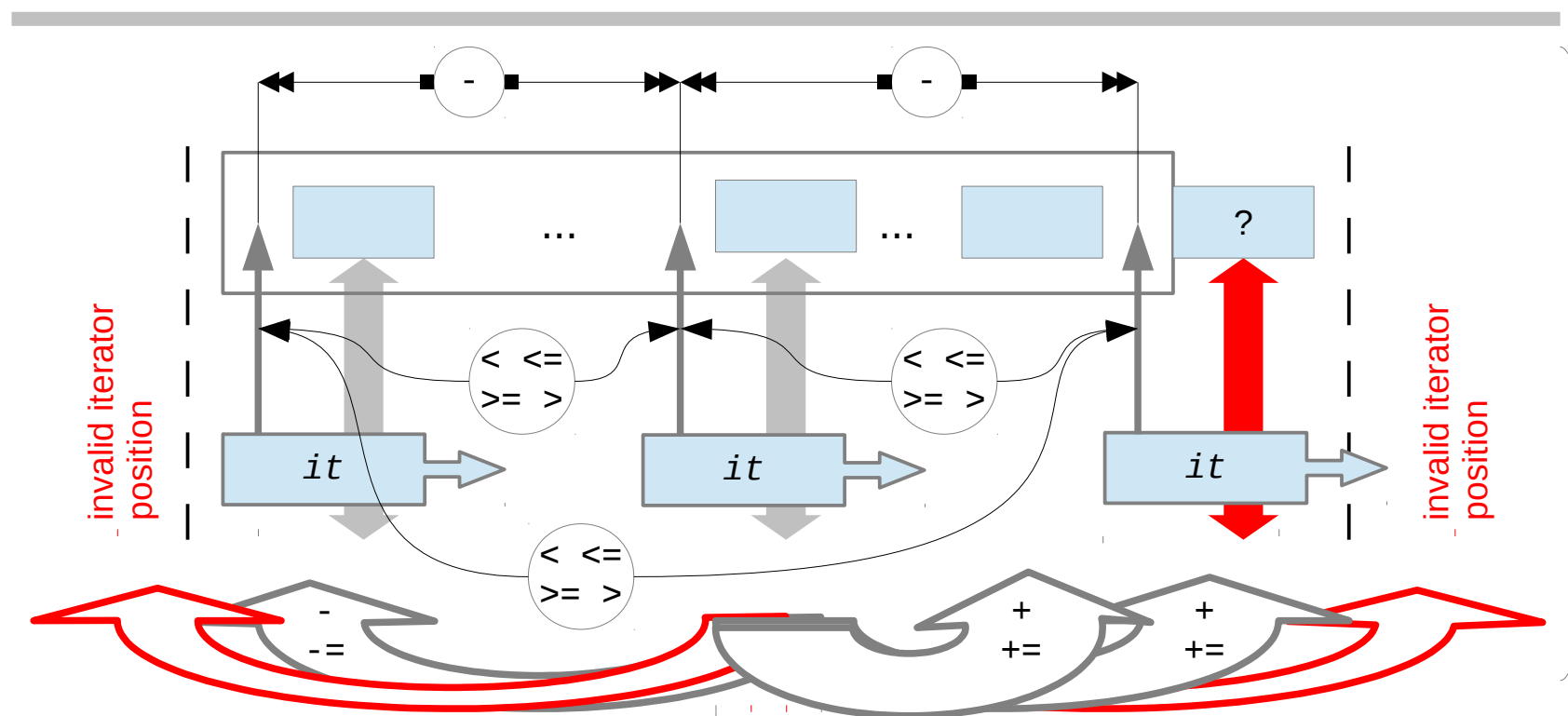
Operations of Unidirectional Iterators

	Effect	Remarks
$*it$	access referenced element	undefined at container end
$++it$ $it++$	advance to next element (usual semantic for pre-/postfix version)	
$it == it$	compare for identical position	operands must denote existing element or end of same container
$it != it$	compare for different position	



Additional Operations of Bidirectional Iterators

	Effect	Remarks
$--it$ $it--$	advance to previous element (usual semantic for pre-/postfix version)	undefined at container begin



Additional Operations of Random Access Iterators

	Effect	Remarks
$it + n$ $it += n$	<i>it</i> advanced by <i>n</i> -th next element (previous if $n < 0$)	resulting iterator position must be inside container (denote existing element or end)
$it - n$ $it -= n$	<i>it</i> advanced by <i>n</i> -th previous element (next if $n < 0$)	
$it - it$	number of increments to reach rhs <i>it</i> from lhs <i>it</i>	operands must denote existing element or end of same container
$it < it$	true lhs <i>it</i> before rhs <i>it</i>	
$it <= it$	true if lhs <i>it</i> not after rhs <i>it</i>	
$it >= it$	true if lhs <i>it</i> not before rhs <i>it</i>	
$it > it$	true if lhs <i>it</i> after rhs <i>it</i>	

STL – Iterator Categories

	Container Dimension												
Library	STL								Standard Strings	Iterator Interface to I/O-Streams		e.g. Boost	Others
Kind of Container	Sequential Containers				Associative Containers								
Data Structure	Random Access		Sequential Access		Tree	Hash	Tree	Hash		I/O operations for some type T			
Class Name	vector	deque	list	forward_list	set	unordered_set	map	unordered_map	string wstring ...	istream_iterator	ostream_iterator		
					multi_set	unordered_multi_set	multi_map	unordered_multi_map					
Iterator Category	Random Access Iterators		Bidirectional Iterators	Unidirectional Iterators	Bidirectional Iterators				Random Access Iterators	Input Iterators	Output Iterators		
Dereferenced Iterator	accesses element						accesses key-value-pair		single character	single item of type T			
	operations available via iterators												

Algorithm Dimension	STL	
	Access:	• find • search • ...
	Modify:	• remove • sort • ...
	Misc:	• count • mimmax • ...
e.g. Boost	Algorithm:	• join • ...
	String_algo:	• trim_left • trim_right • ...
Others		

operations expected from iterators



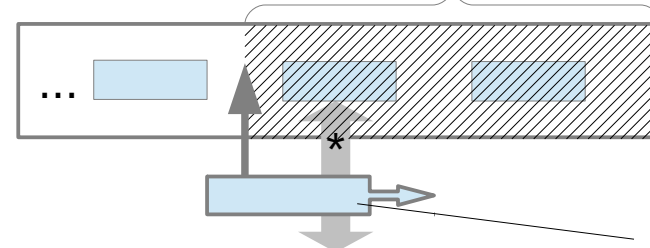
Failure to comply will cause a compile time error, typically with respect to the header file that defines the algorithm.

Iterators as "Glue" to connect Containers with Algorithms



Failure to comply will either cause a compile time error or show at runtime and may depend on the kind of container.

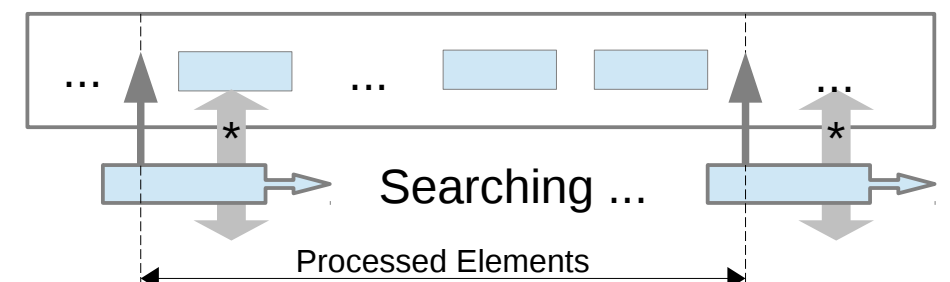
elements still physically present though no longer logically part of the container



"Removing" Elements ... Return "New End"

Use of iterators to specify container elements to process:

- starting point is the first element to process
- ending point is the first element **not** to process
- whole container is specified via its begin() and end()

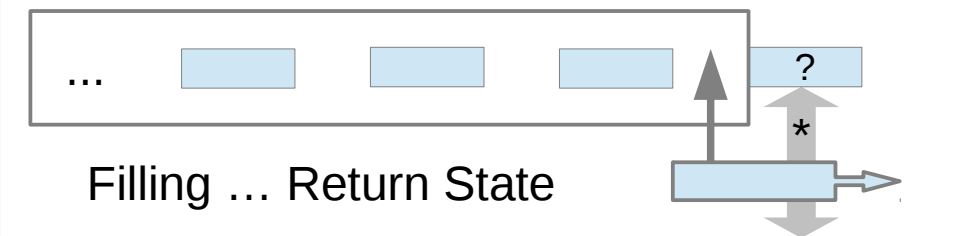


always valid for dereferencing

... Return Success ...

not necessarily valid for dereferencing!

... or Failure



Input Iterators Semantic Restrictions

- * must only be used for read access
- ++ must follow each read exactly once

Output Iterators Semantic Restrictions

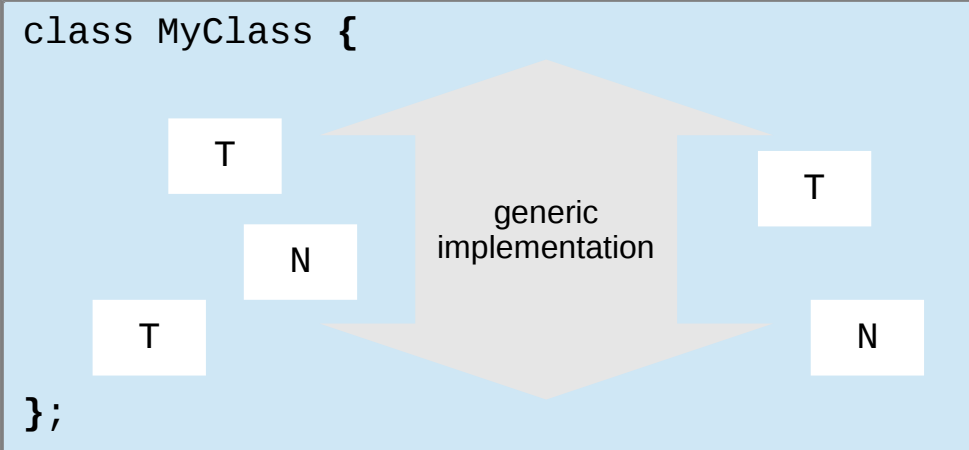
- * must only be used for write access
- ++ must follow each write exactly once

STL – Iterator Usages

keywords class and typename have the same meaning in template parameter lists

Template Class

```
template<typename T, int N>
```

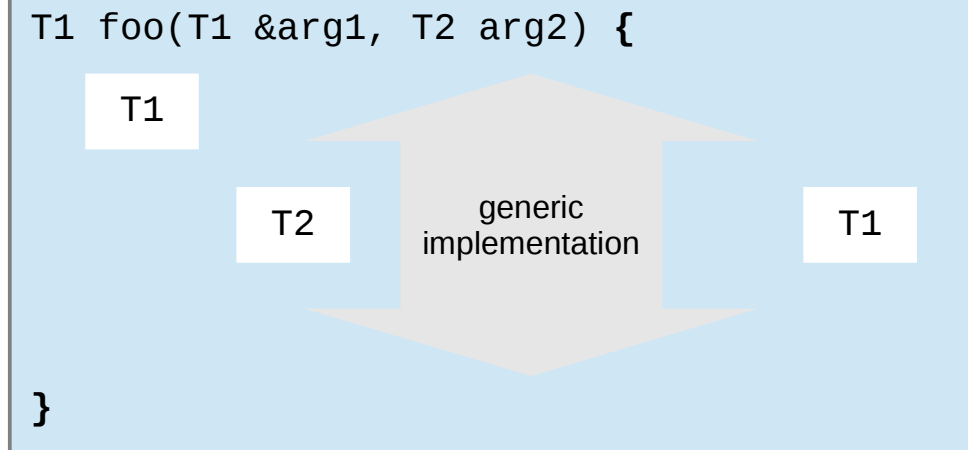


types and constants may be parameterized

preliminary
syntax
checking

Template Function

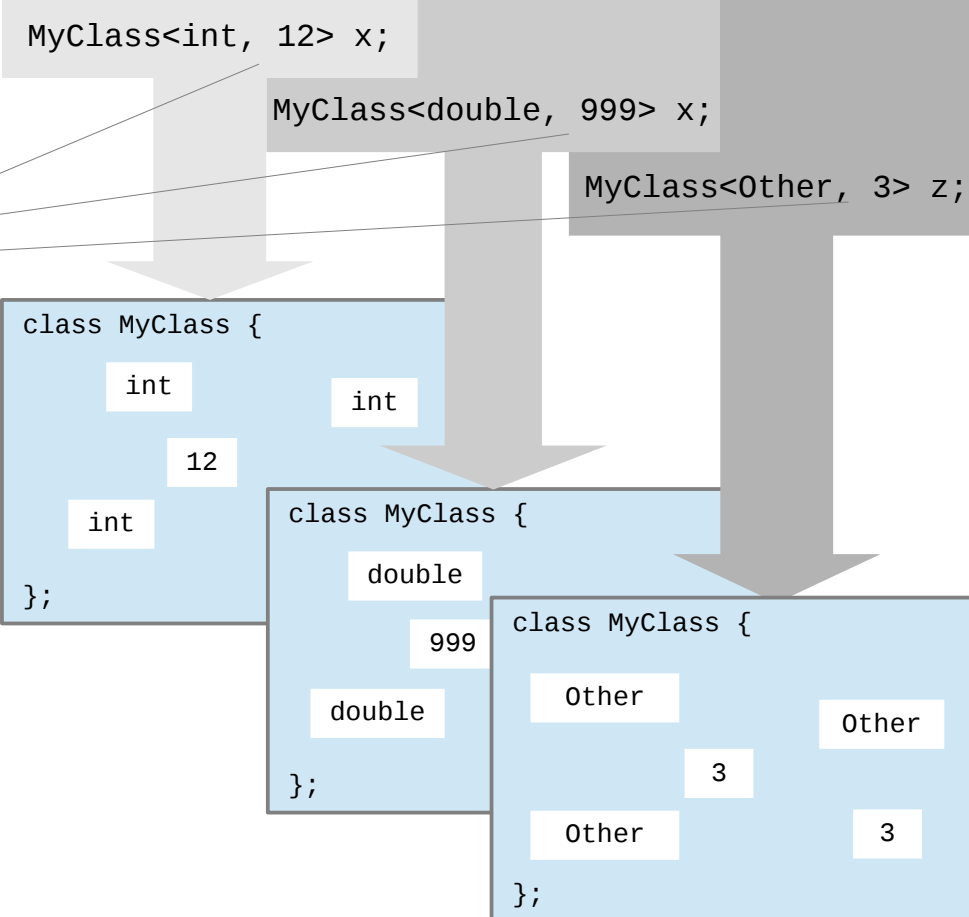
```
template<typename T1, typename T2>
```



typically only types are parameterized

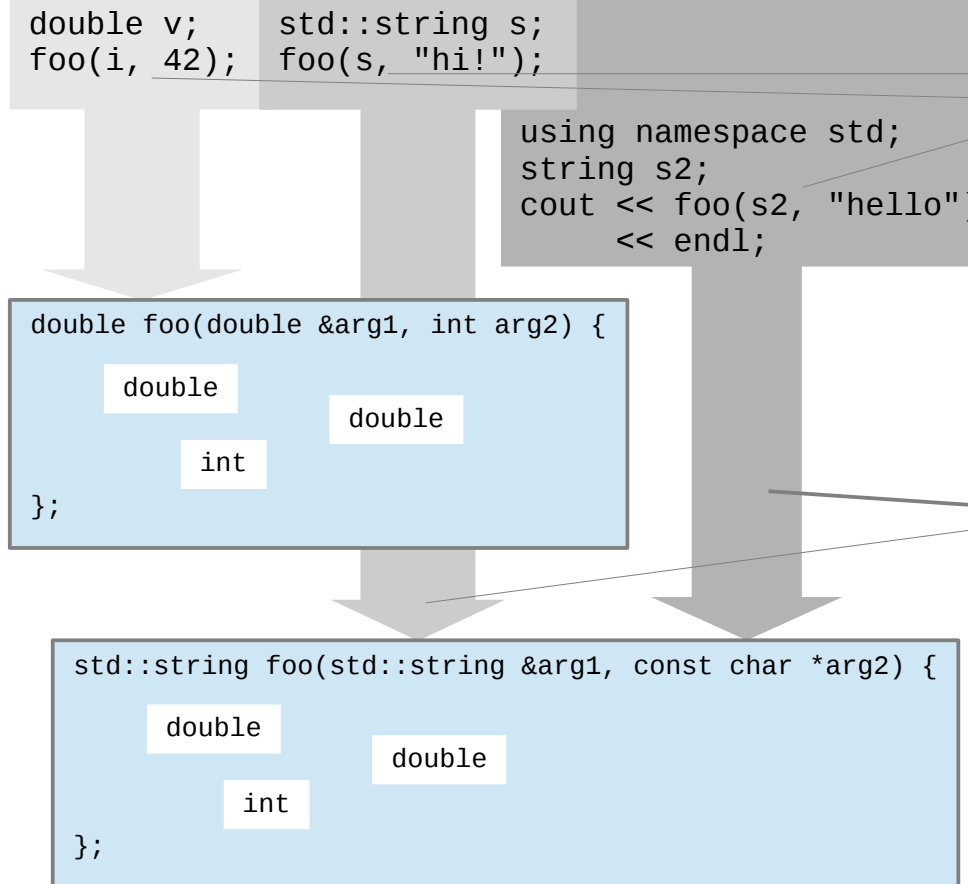
Template definition extends to end of block (i.e. class or function body)

Compiler-Dependant Intermediate Representation



for template classes type and value arguments must **always** be supplied

template-aware
code
generation



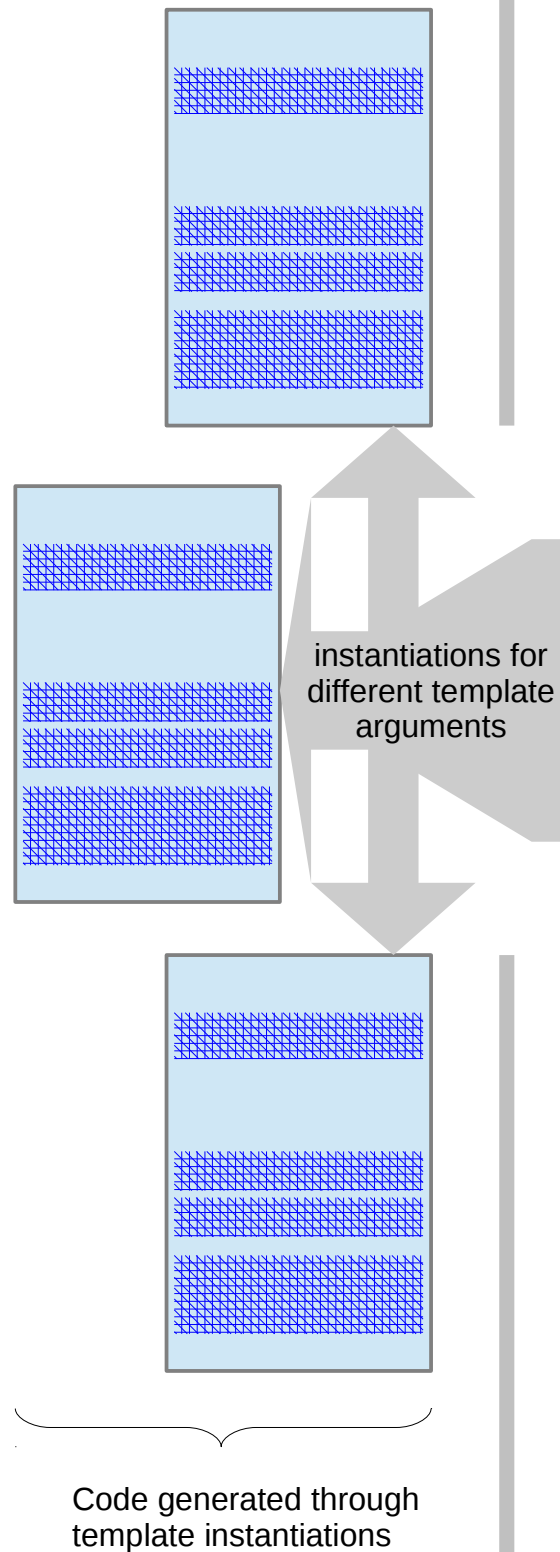
for template functions types are typically deduced at the call site

duplicated non-inline versions of functions (with identical set of instantiation types) are usually "optimized out" at link time

Code Compiled and Optimised for Specific Template Arguments

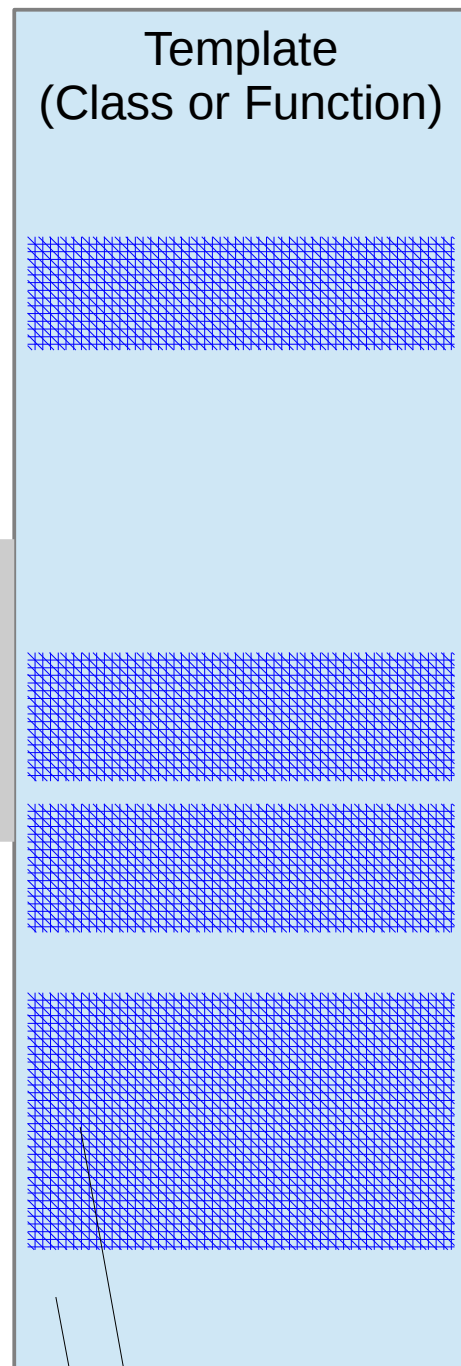
Template Basics

Code Bloat Risk



Code sections not depending on template arguments generated again and again for each instantiation.

Initial Version

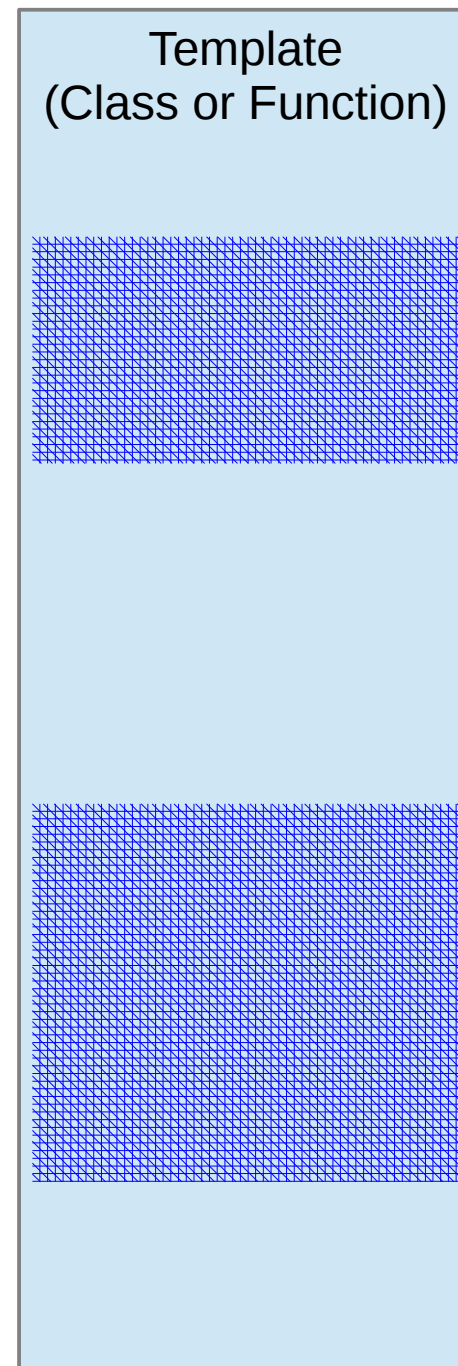


generated code actually depends on template arguments

generated code does not depend on template arguments

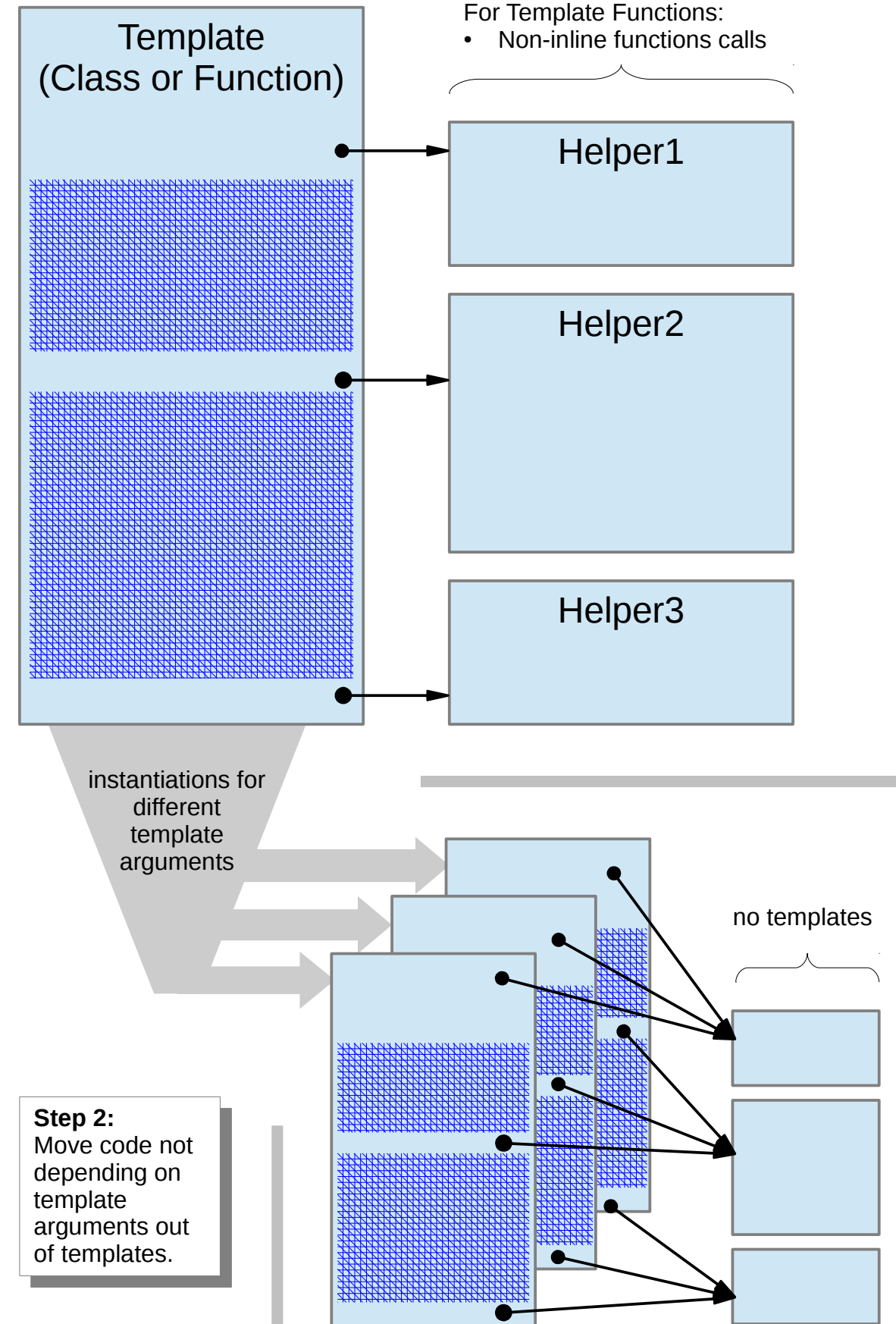
Source code with mixed parts depending and not depending on template arguments.

Intermediate Version



Step1:
Where possible restructure code to concentrate parts depending and parts not depending on template arguments.

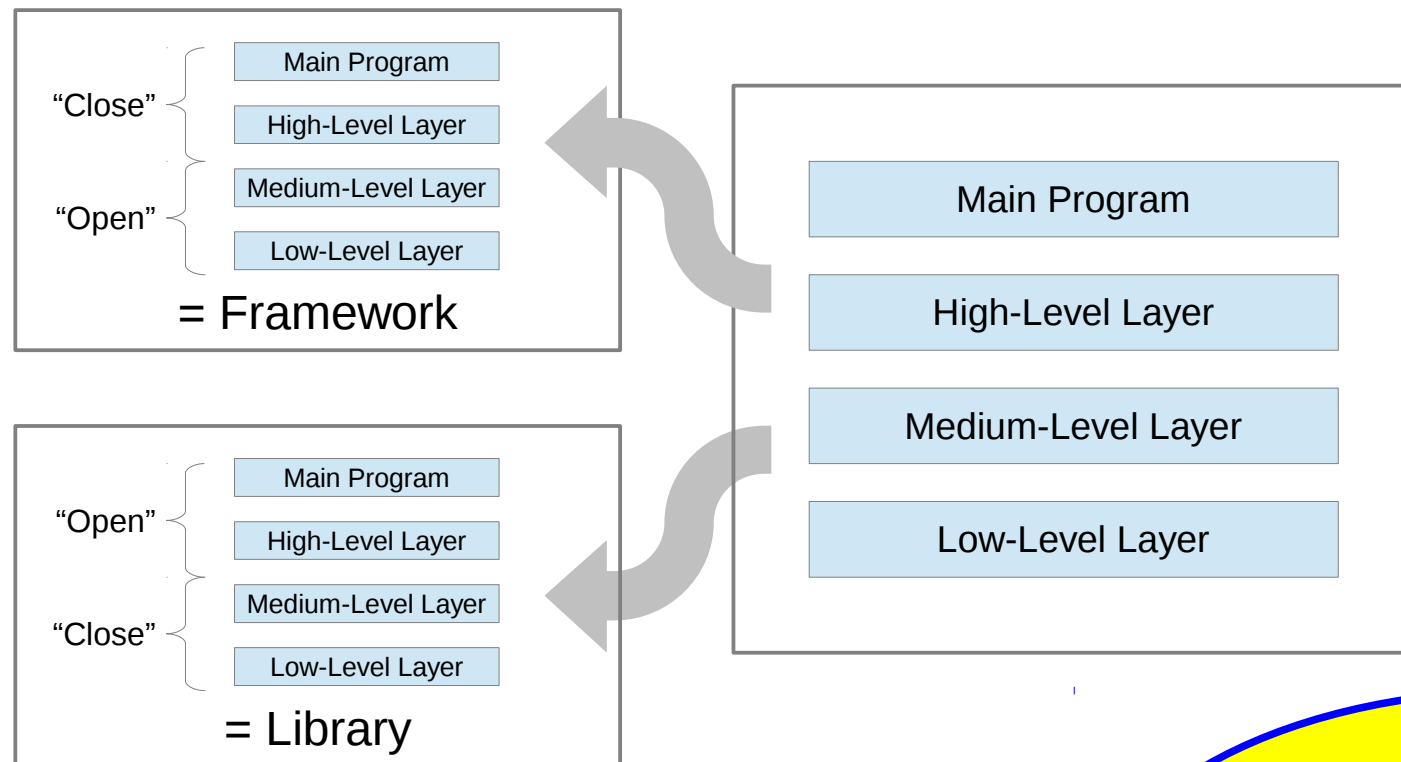
Improved Final Version



Step 2:
Move code not depending on template arguments out of templates.

Code sections not depending on template arguments not any more in templates.

Reducing Code Bloat



Design for Reusability:

- *Libraries* or *Frameworks* for common components
- Classes for common services or abstractions
- C++-Templates for genericity in types

Use Available Tools and Libraries e.g.

- *Doxygen* (or similar) to create good-looking documentation from embedded comments
- The *Boost Platform* for a extremely rich choice of "what seems to be missing or forgotten" in the C/C++ Standard Library

Pick the Best from Agility, at least

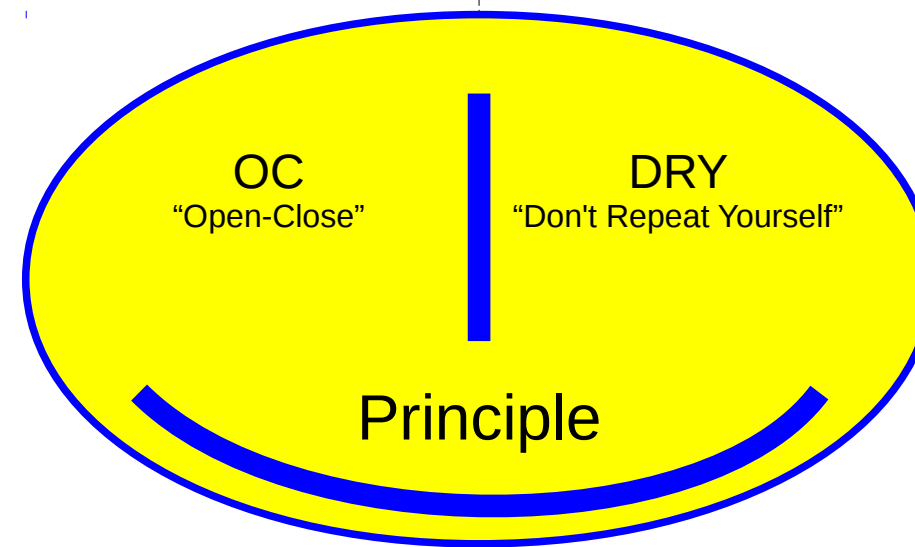
- integrate continuously
- automate boring tests
- (maybe try "pair-programming"?)

Parameterize for Flexibility with

- Run-Time arguments for functions and subroutines
- Compile-Time arguments for templates

Apply "Best Practices" e.g.:

- Standard Design Patterns (from GoF) like
 - Composite
 - Template Methode
 - ...
- Well-known C++ Idioms like
 - PIMPL (Pointer to Implementation)
 - RAIL (Ressource Acquisition is Initialisation)
 - CRTP (Curiously Recurring Template Pattern)
 - ...
- Handy Little Techniques where useful
 - "Named Argument" (from C++ FAQ)
 - "Safe delete" (from Boost)
 - ...



Consider to Write Your Own Tools, e.g. to

- create a C/C++ header file from a spreadsheet or vice versa
- create a CSV- or XML-document from a source file, or even
- create both, source code and auxilliary documents from a DSL (domain specific language)

But always judiciously decide ... and Don't Overdo!

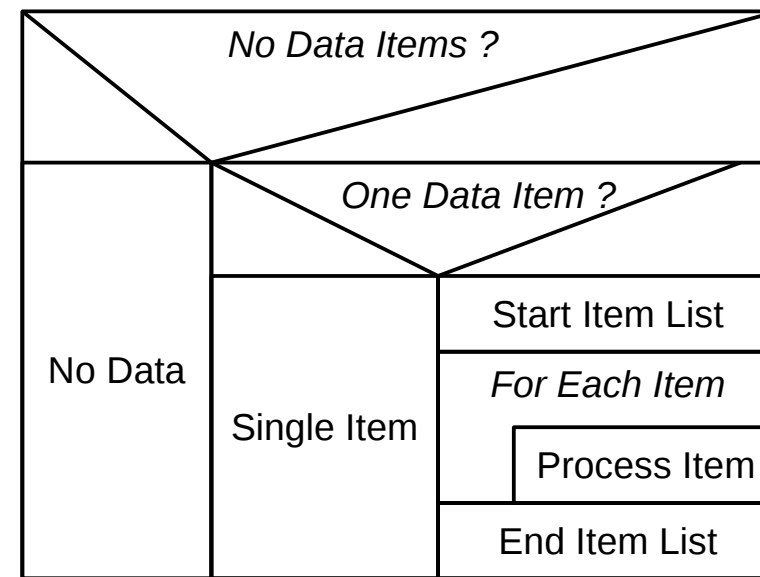
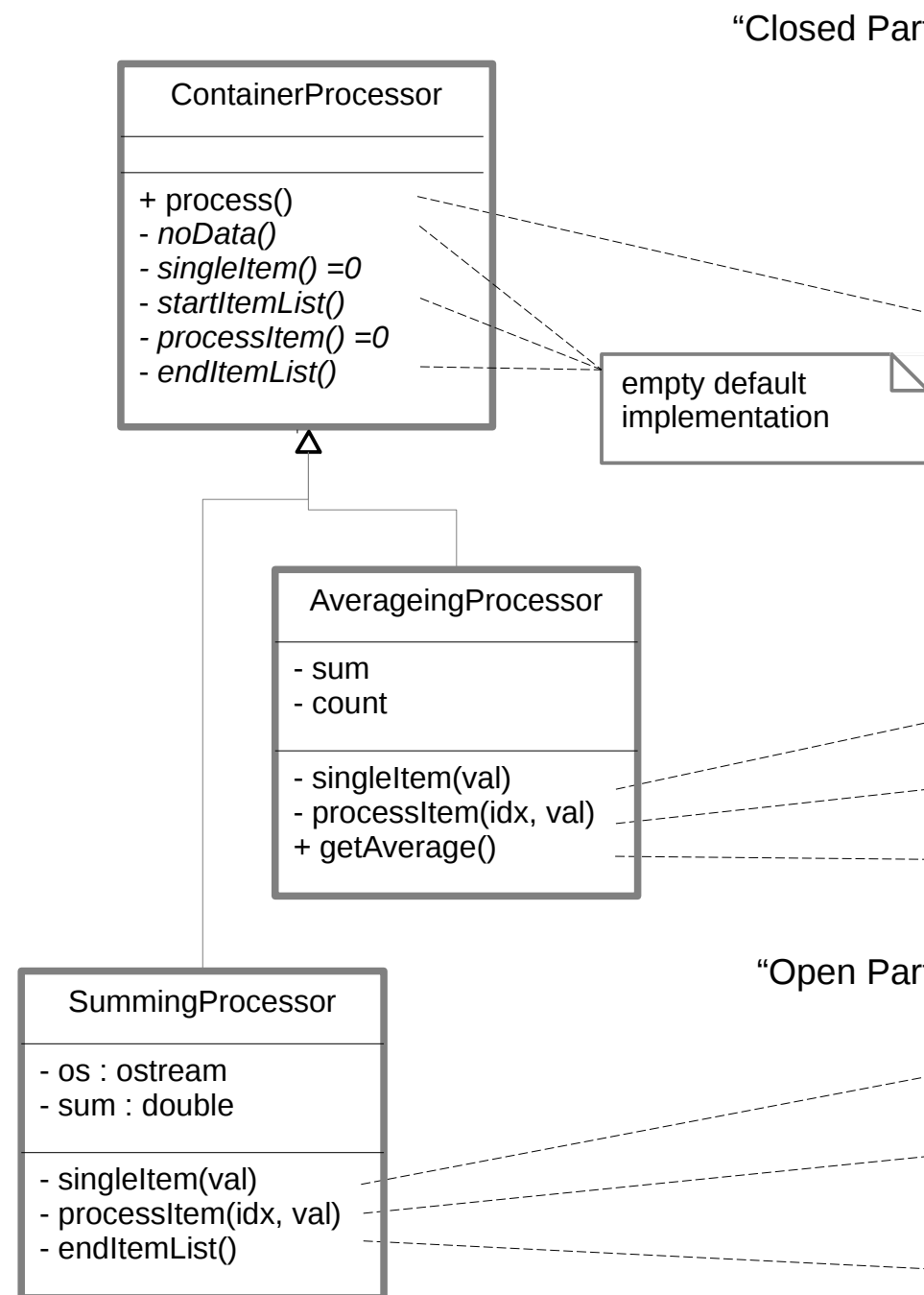
- Not each and every global variable needs to be turned into a **Singleton**.
- Not each and every little config file needs to be parsed as full **XML**.
- Not each and every small class needs type genericity.
- ...

If you can't avoid a complex design in the end, at least provide some easy to use defaults for the most common use cases!

Guiding Principles

Implementation Based on Virtual Member Functions

As described under the entry “Template Methode”-Pattern in the GoF-book. Standard technique in all OO programming languages that support polymorphism but not type-generic programming.



```

if (data.empty())
    noData();
else if (data.size() == 1)
    singleItem(data.at(0));
else {
    startItemList();
    int n = 0;
    for (auto value : data)
        process(item(++n, value));
    endItemList(n);
}
  
```

“Open Part”

```

sum = val;
count = 1;

sum = +val;
count = idx;

return sum / count;

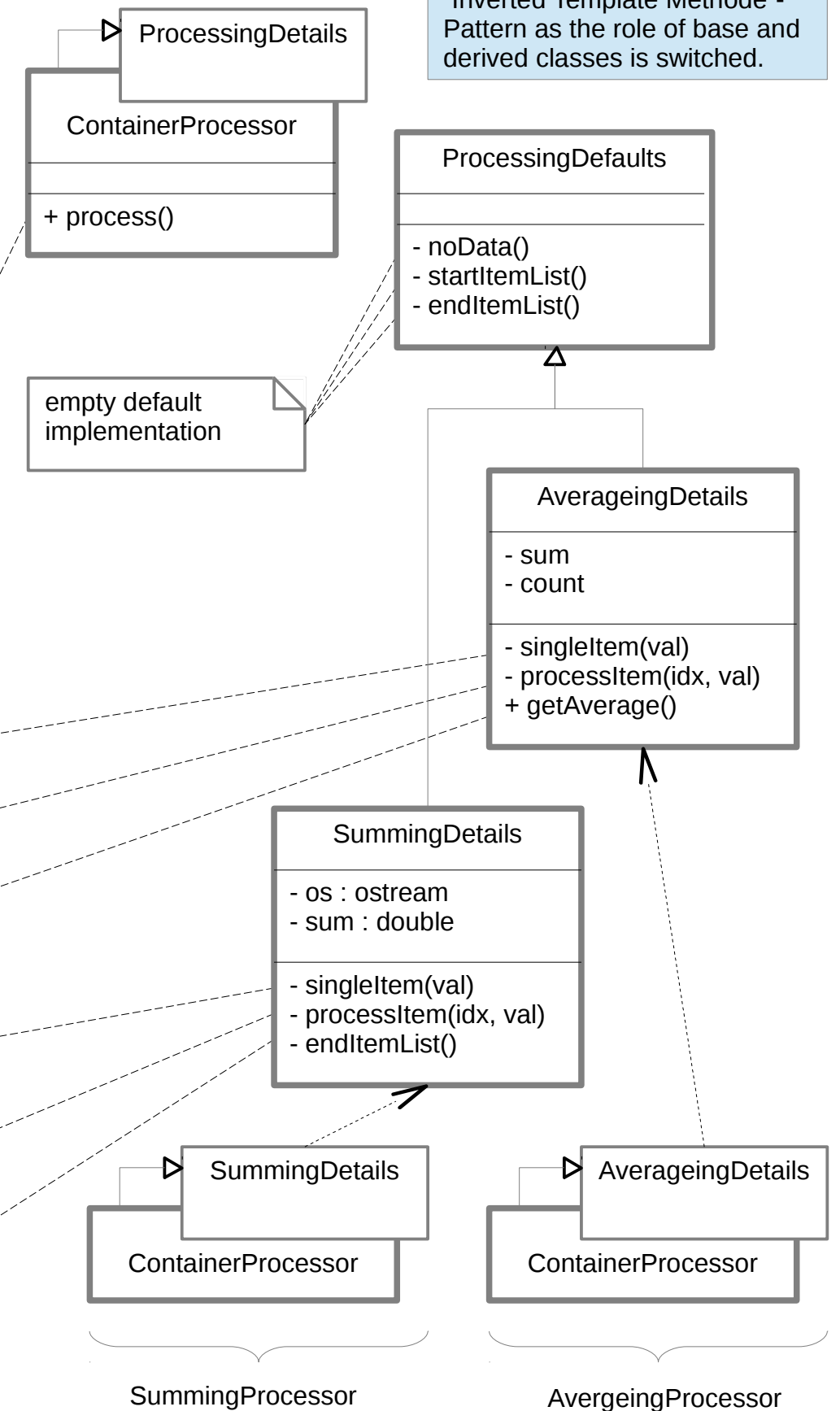
os << val;

os << setw(3) << idx << ':'
  << setw(8) << val << '\n';
sum += val;

os << "-----\n"
  << setw(11) << sum << '\n';
  
```

Implementation Based on the C++ Template Mechanism

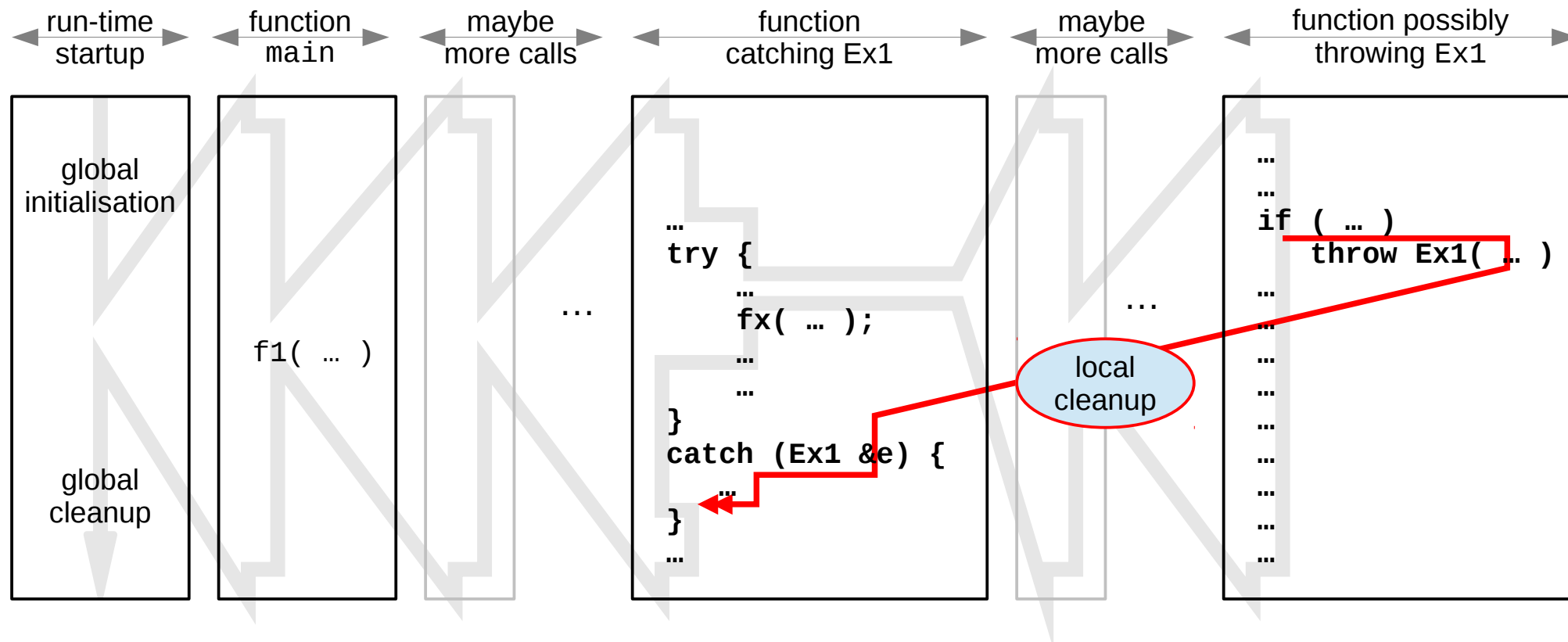
Sometime also named “Inverted Template Methode”-Pattern as the role of base and derived classes is switched.



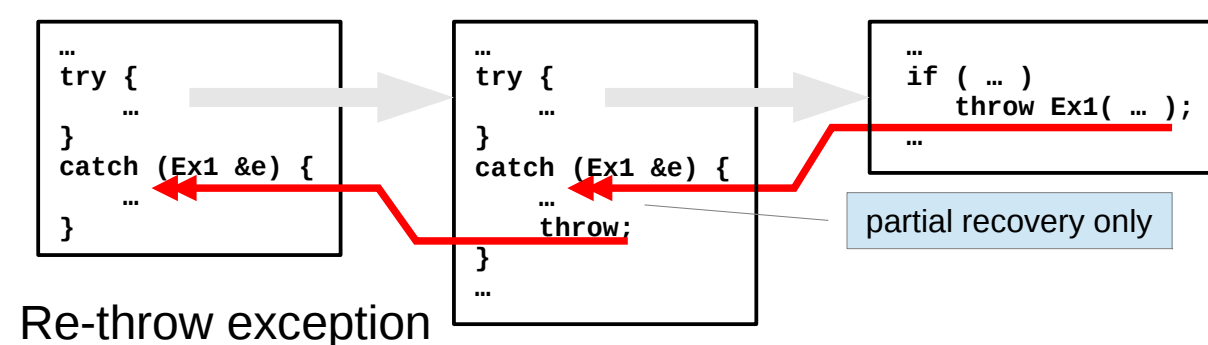
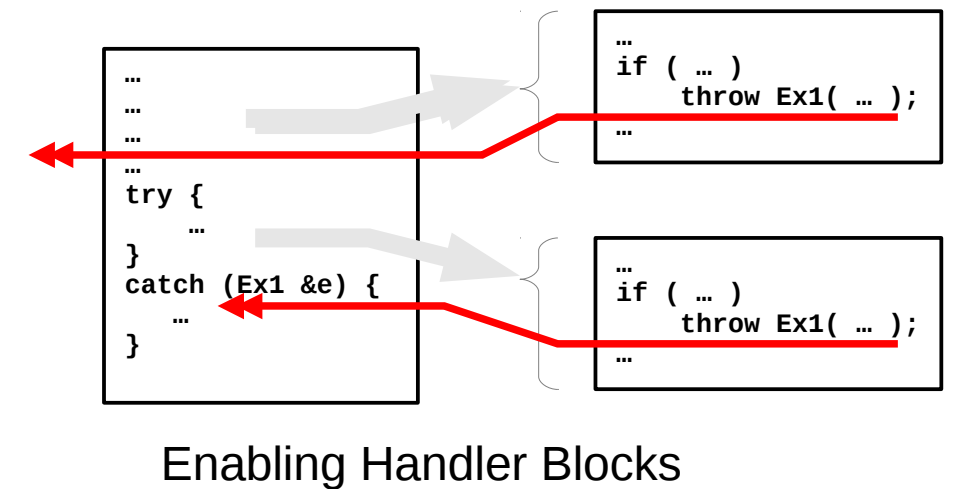
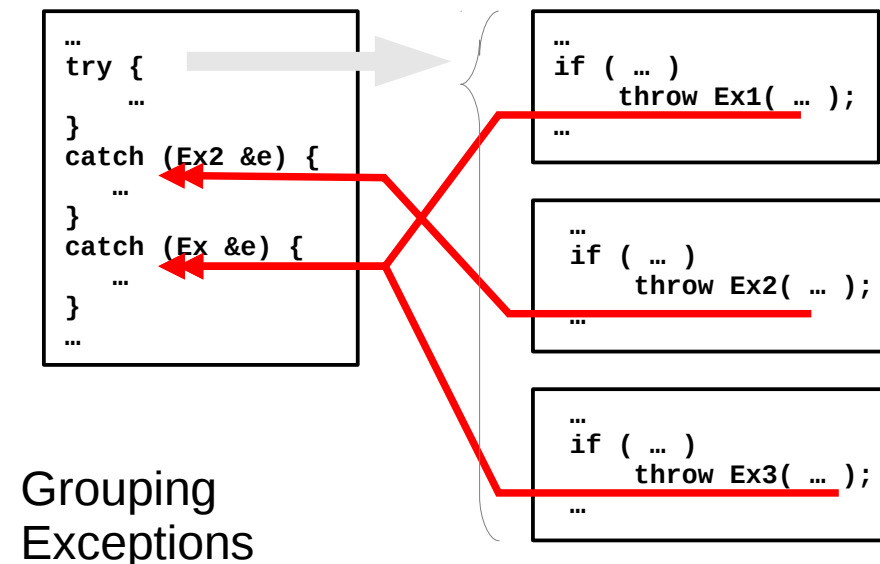
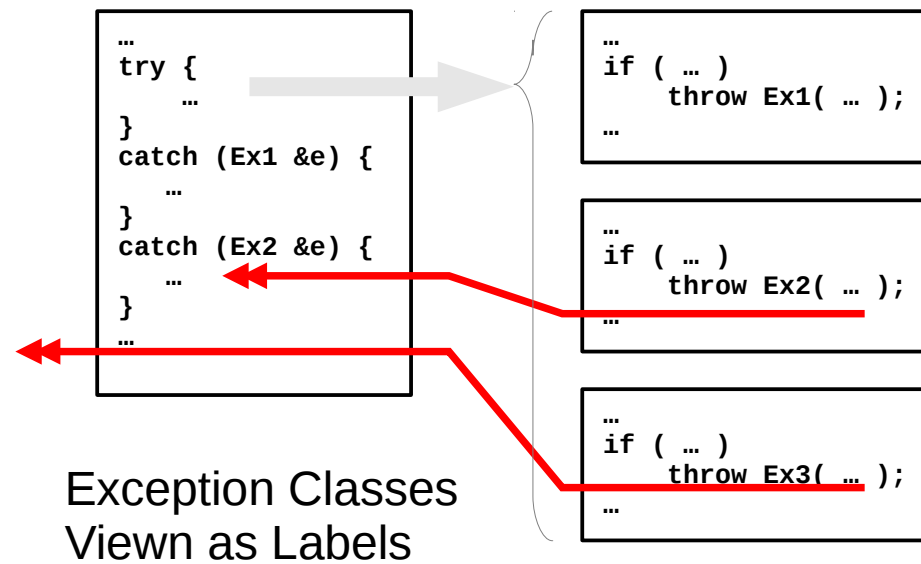
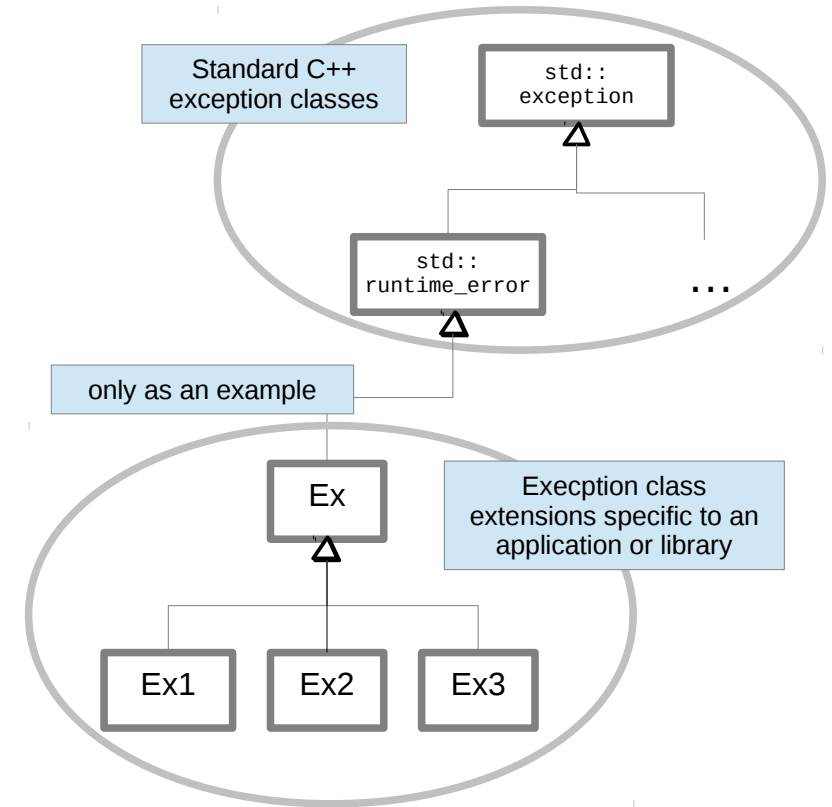
Example – “Open Close”-Principle

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

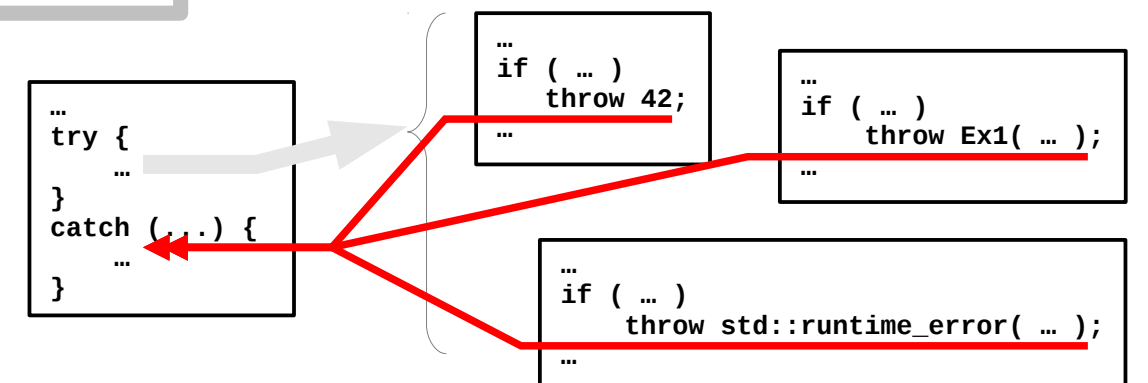
Execution Path taken for Exception



Exception Class Hierarchies

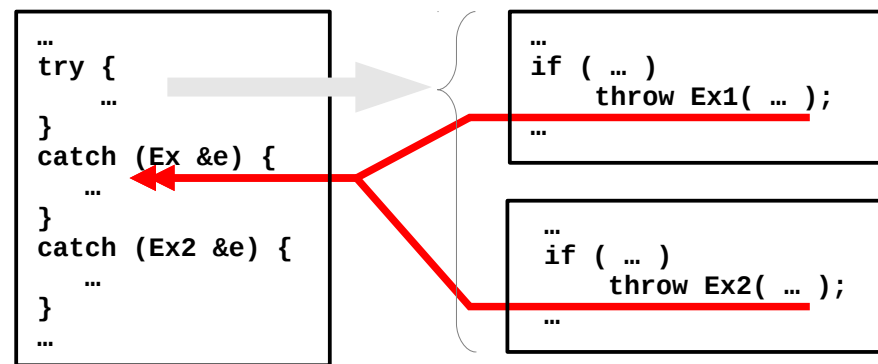


Catch Any Exception



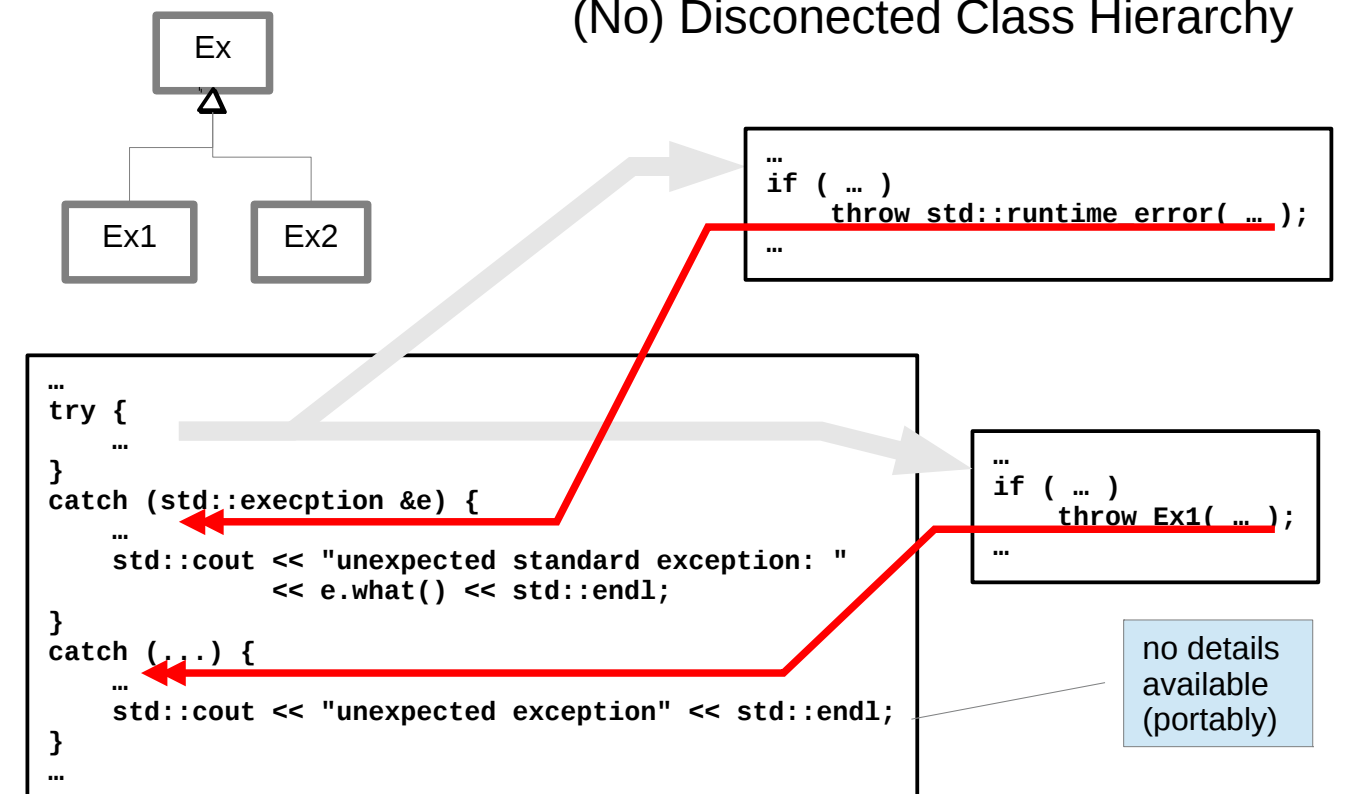
Excoption Basics

(Bad) Order of Handlers

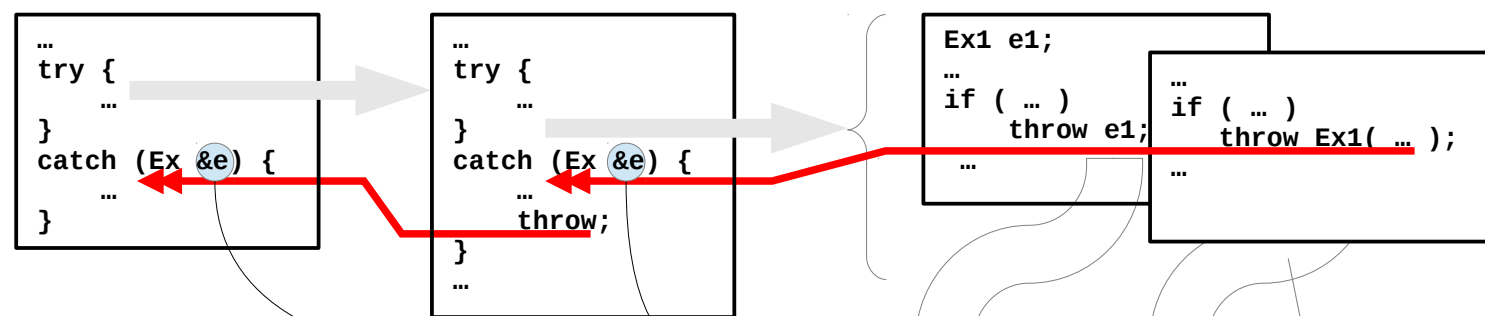


The compiler may issue a warning that the second catch-clause is shadowed by the first but this is not mandatory.

(No) Disconnected Class Hierarchy



Optimal Re-throwing



by reference
Memory location guaranteed to exist until last catch-block accessing exception object

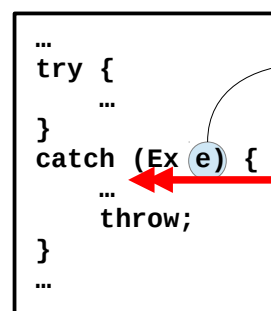
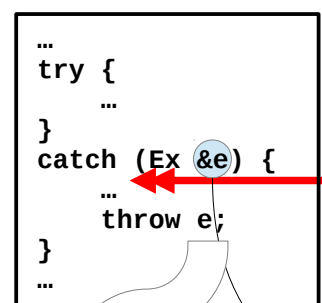
Ex1 object formally created with copy-constructor

may use RVO

physical copy, no polymorphism

Memory location in stack-frame of function containing try-catch-block

Slicing Exception Object



by value

Ex object created by slicing Ex1

Ex object created as copy, thereby possibly sliced

Memory location from rethrowing, guaranteed to exist until last catch-block

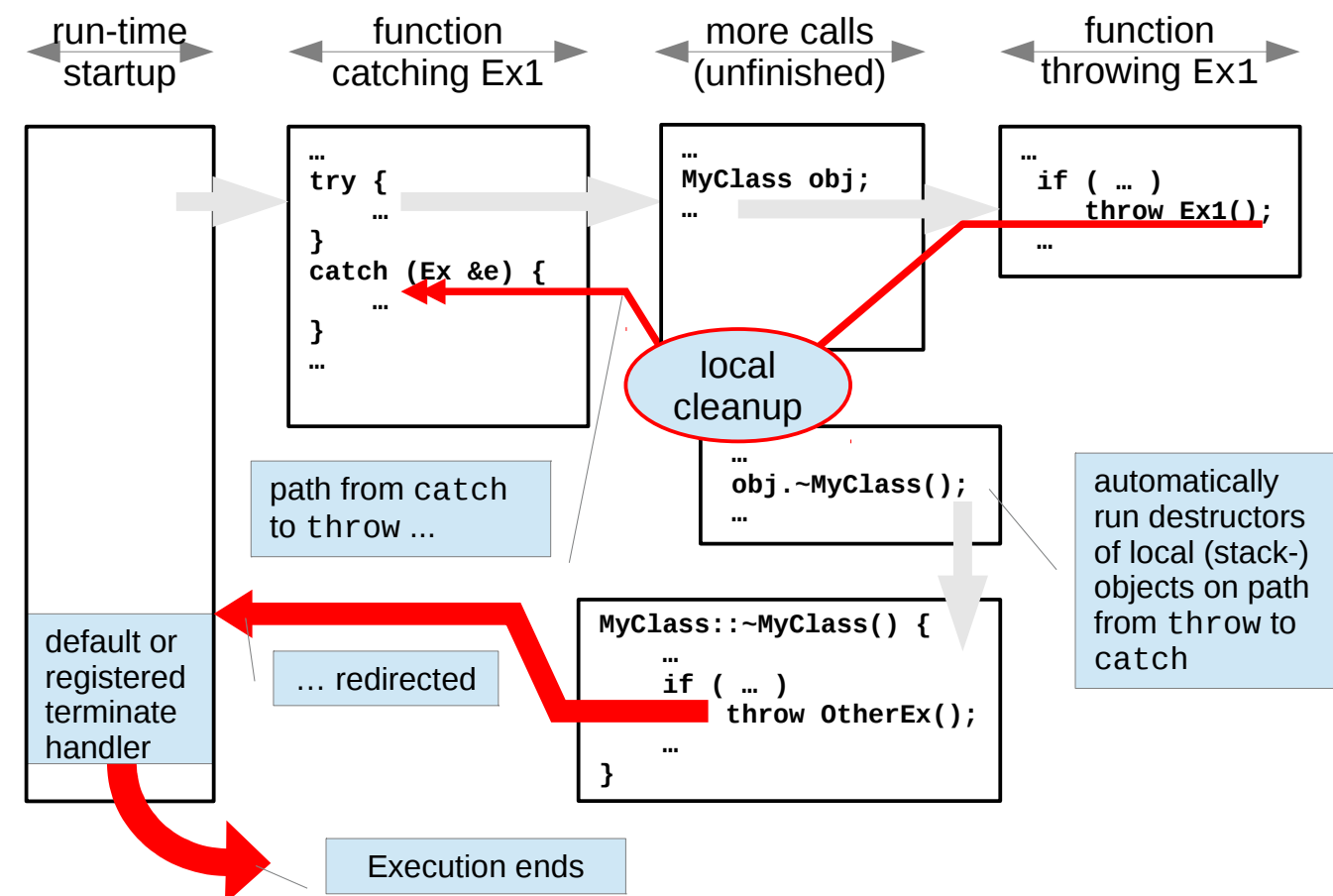
Object of derived class Ex1 or Ex2

Memory location from initial throw, guaranteed to exist until current catch-block ends

Sub-optimal Re-throwing

Exception Details

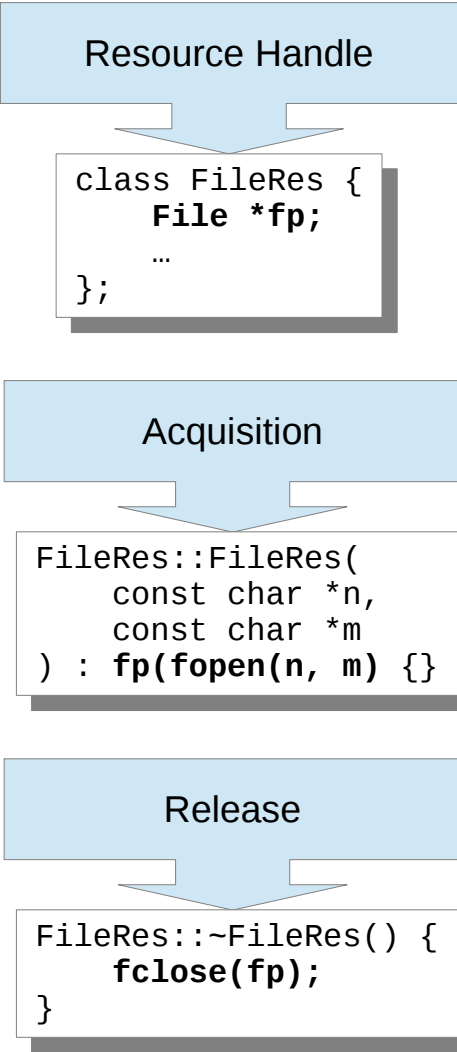
(No) Throwing from Destructors



Classic Resource Management APIs

Turn into RAI

Principles	Examples					
	Unix/Linux		C	C Free Memory (Heap)		C++11
	Processes	Files	Files	C++ Free Memory (Heap)		<code>std::mutex m;</code>
Operation to acquire returns ...	<code>fork()</code>	<code>creat()</code> , <code>open()</code>	<code>fopen()</code> , <code>freopen()</code>	<code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code>		<code>m.lock()</code> , <code>m.try_lock()</code>
				<code>new T</code>	<code>new T[N]</code>	
... some handle to identify resource ...	<code>pid_t</code> (some integer)	<code>int</code>	<code>FILE *</code> (pointer to some struct with opaque content)	generic pointer (<code>void*</code>) to otherwise unused storage for (at least) as many bytes as requested		no special return value (instead state of object is changed)
				<code>T*</code> denoting a pointer to otherwise unused storage for (at least) one object of type <code>T</code>	<code>T*</code> denoting a pointer to otherwise unused storage for (at least) <code>N</code> objects of type <code>T</code> at adjacent memory locations like in a builtin array	
... in subsequent operations like ...	<code>kill()</code> , <code>ptrace()</code> , ...	<code>read()</code> , <code>write()</code> , <code>seek()</code> , <code>poll()</code> , ...	<code>fread()</code> , <code>fwrite()</code> , <code>fseek()</code> , <code>ftell()</code> , <code>fflush()</code> , ...	after conversion to the target type all builtin pointer operations		<code>m.native_handle()</code>
				all builtin pointer operations		
... until final release (eventually returning resource to a pool)	<code>wait()</code> , <code>waitpid()</code>	<code>close()</code>	<code>fclose()</code>	<code>free()</code>		<code>m.unlock()</code>
				<code>delete ...</code>	<code>delete[] ...</code>	
Standard Wrapper	none	none	none	<code>std::unique_ptr<T></code>	<code>std::unique_ptr<T[]></code>	<code>std::lock_guard</code>



Acquire Resource for Code Segment

```
...  
{  
    FileRes f( ... );  
    ...  
    if ( ... )  
        return;  
    ...  
    foo( ... );  
    ...  
    if ( ... )  
        break;  
...  
}
```

void foo() {
 ...
 if (...)
 throw ...;
...
}

Acquire Resource for Object Lifetime

```
class MyClass {  
    ...  
    FileRes fr;  
public:  
    MyClass( ... )  
        : fr( ... )  
    { ... }  
};
```

```
FileRes f( ... );  
...  
char s[80];  
fgets(s, sizeof s, f);  
...  
if (!ferror(f))  
    ...
```

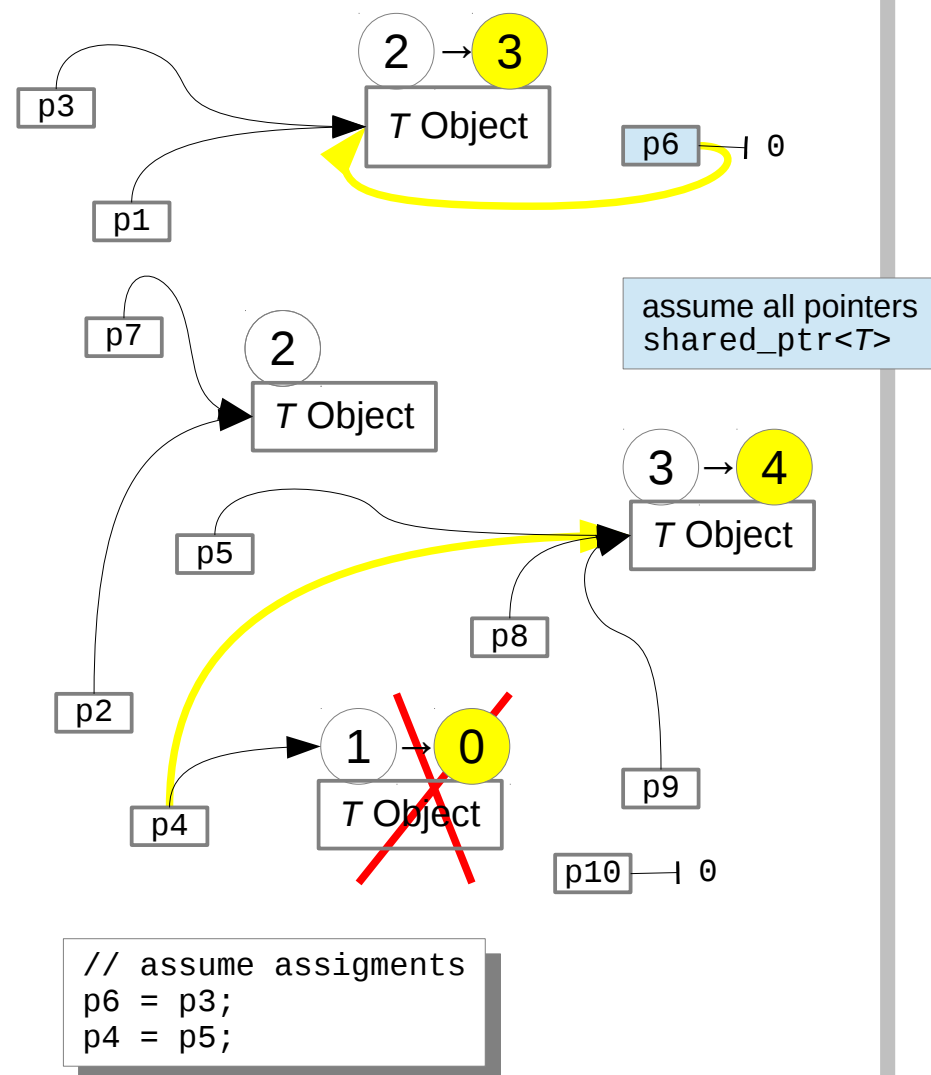
Optionally add Convenience Operations

```
bool FileRes::is_open() const {  
    return (fp != nullptr);  
}
```

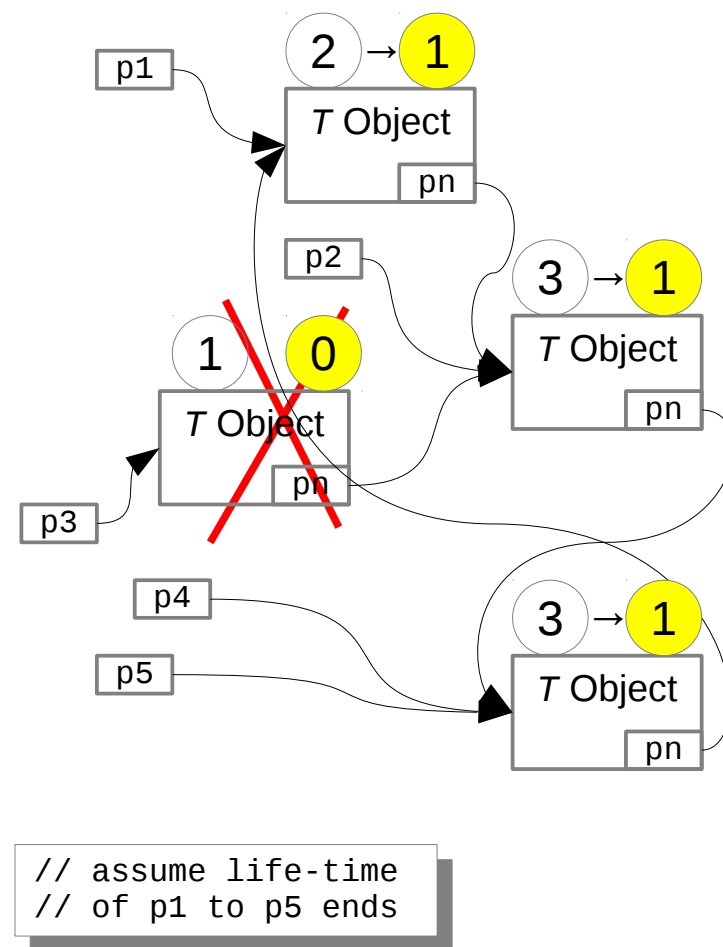
Easy and Secure Use via Automatic Conversion

```
FileRes::operator File*() {  
    if (!is_open())  
        throw std::runtime_error("not open");  
    return fp;  
}
```

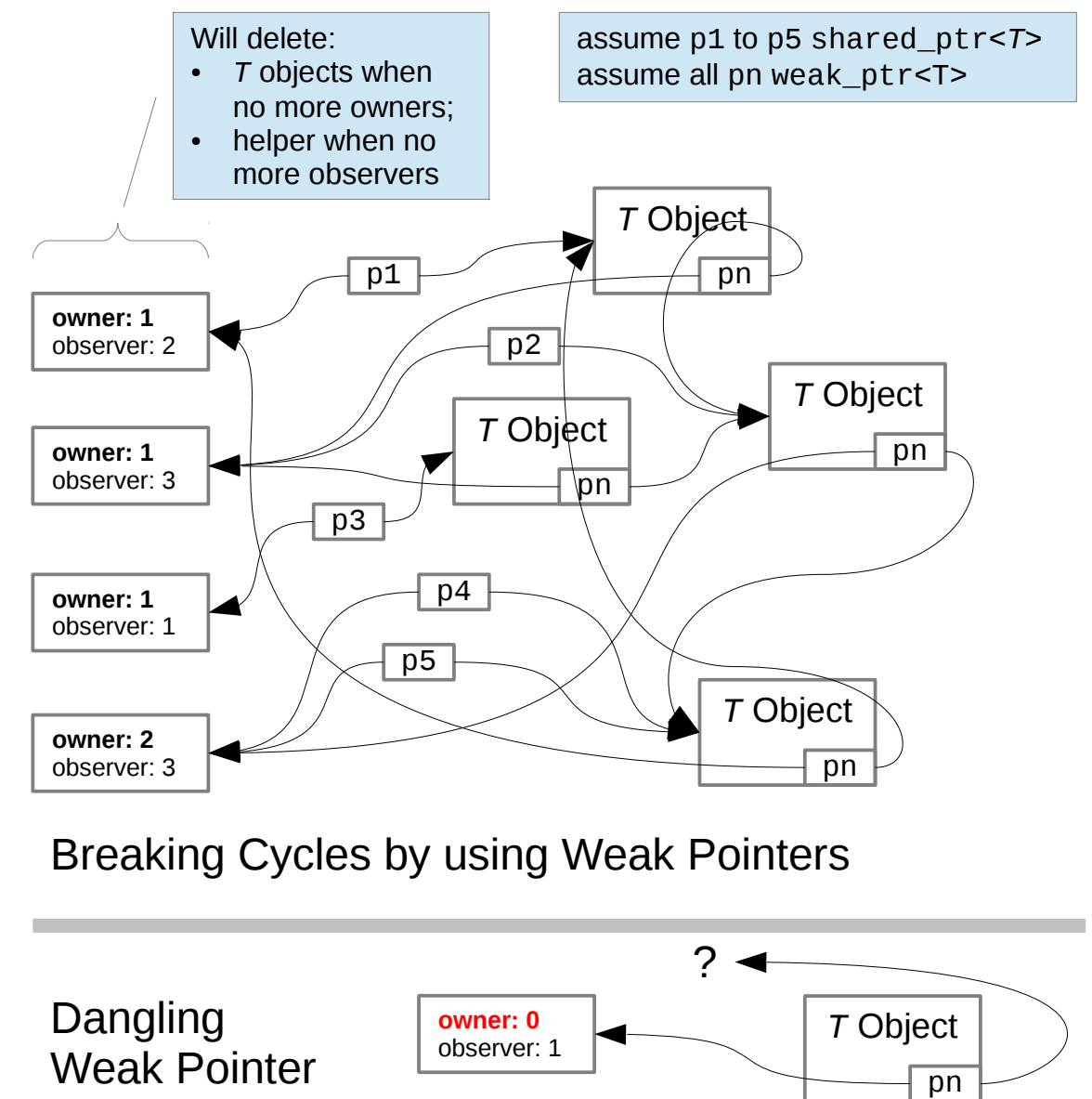
Wrapped Resource



Reference Counting Principle



Problem of Cyclic References



Comparing ...	<code>std::unique_ptr<T></code>	<code>std::shared_ptr<T></code>	Remarks
Characteristic	refers to a single object of type <code>T</code> , uniquely owned	refers to a single object of type <code>T</code> , possibly shared with other referrers	may also refer to "no object" (like a <code>nullptr</code>)
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 <code>T</code> destructor must also be called in an assignment if the current referrer is the last 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	

*: explicit use of `std::move` for argument is a possible work-around

Smart Pointer Comparison

(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, <http://tbfe.de>

