Top Left

Top

Top Right

Enjoy the White Nights

LEFT

Wild Wild West

The Classical Methods

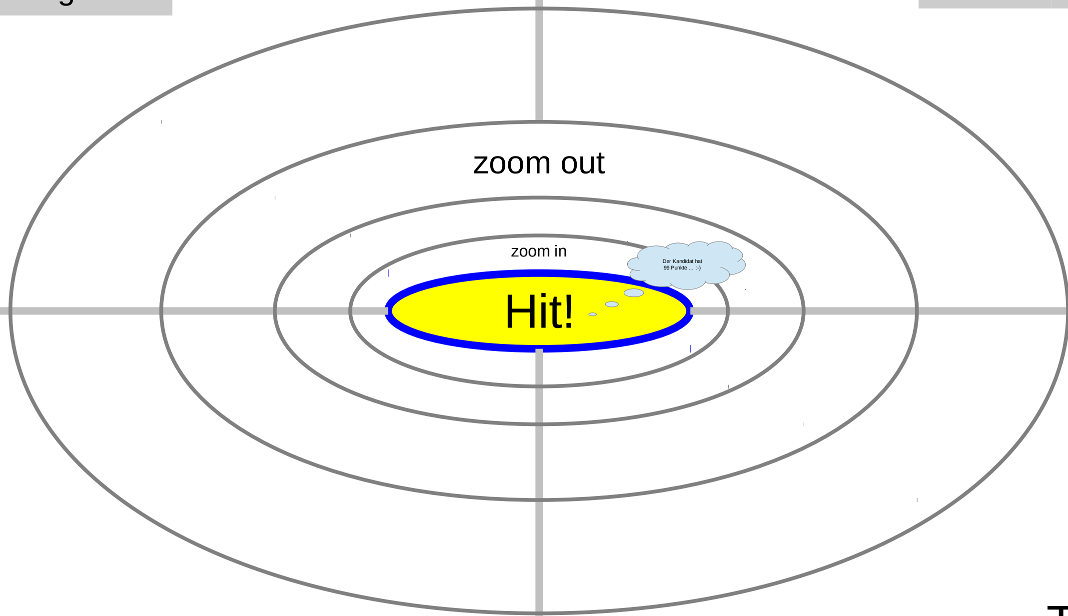
View-Menu	**Zoom**	in (closer)
		out
Right Scrollbar	**Vertical shift**	to top
		to bottom
Bottom Scrollbar	**Horizontal shift**	to left
		to right

More to the Bottom

Scroll-Wheel Navigation

CTRL	forward	**Zoom**	in (closer)
	backward		out
(none)	forward	**Vertical shift**	to top
	backward		to bottom
SHIFT	forward	**Horizontal shift**	to left
	backward		to right

More to the Right



More to the Left

Eastern Wisdom

RIGHT

Keyboard Shortcuts

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

More to the Top

Touch-Sreen Navigation

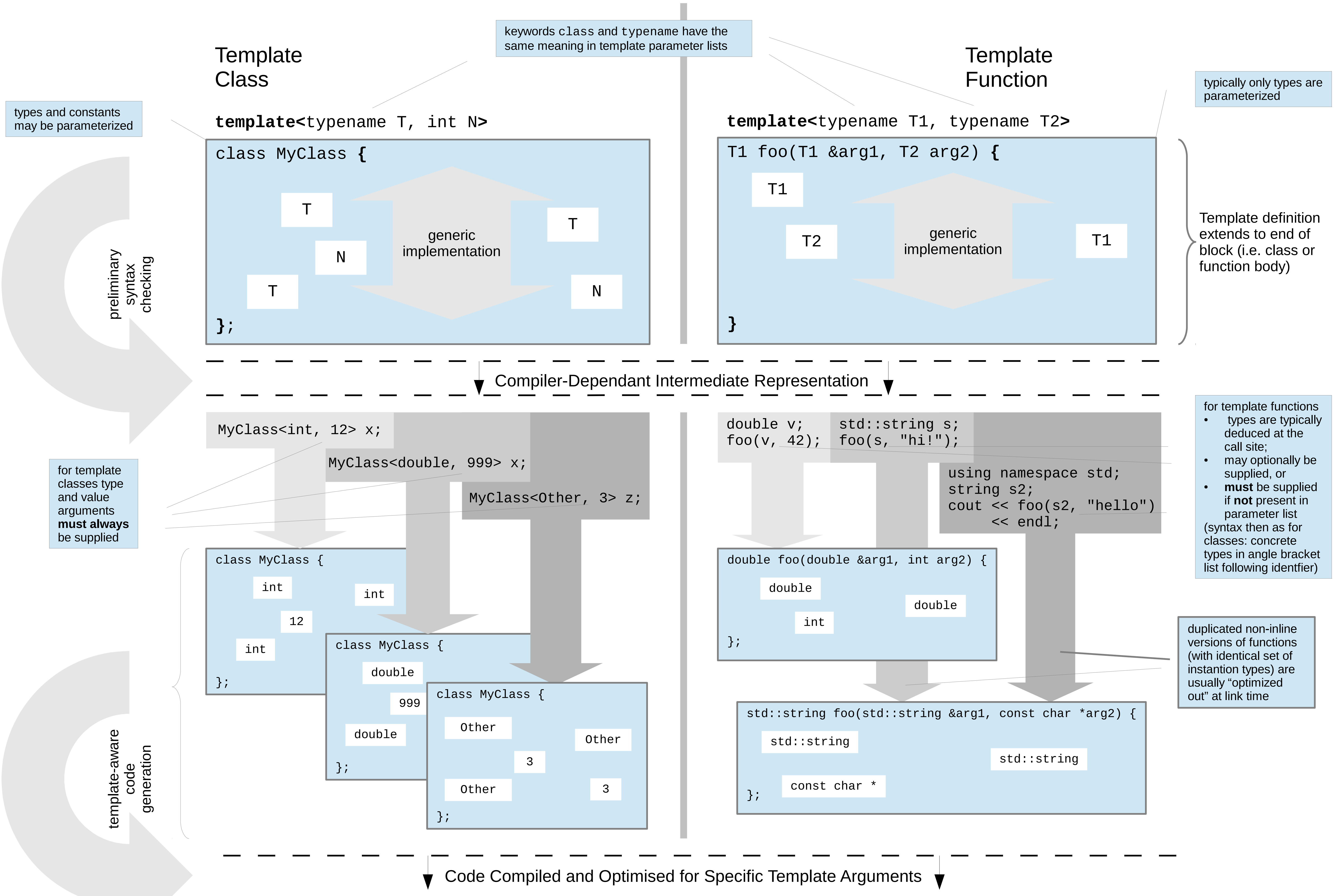
try the usual guestures (as this depends on the **browser** or **viewer** and on the **device** it may or may not work)	**Zoom**	in (closer)
		out
	Vertical shift	to top
		to bottom
	Horizontal shift	to left
		to right

Beware of the Penguins

Bottom Left

Test Browser Navigation

Bottom Right



Template Basics

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)];
}
```

```
RingBuffer b;
```

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;
RingBuffer<MyClass> z;
```

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.

Parametrizing *Size*

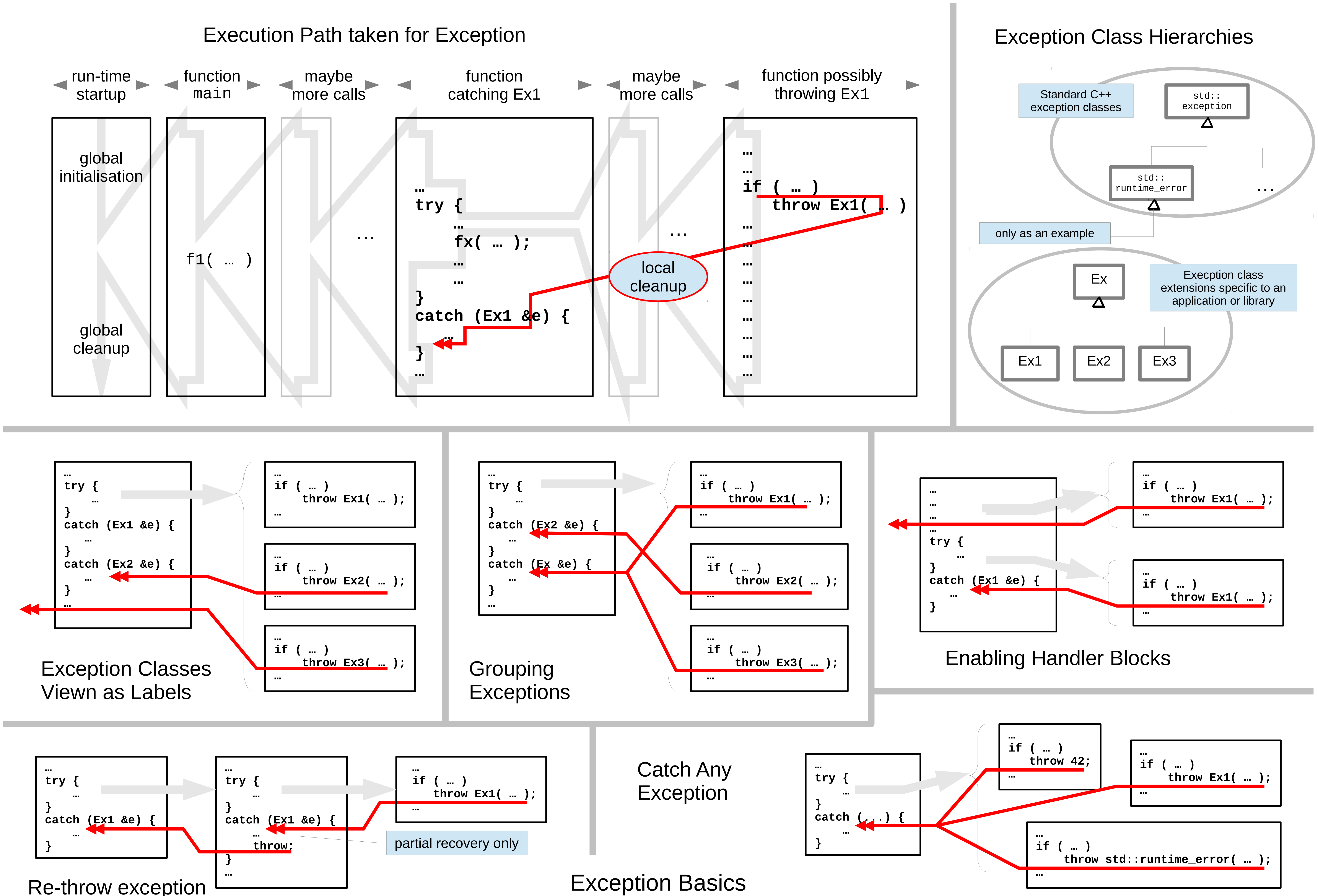
```
template<std::size_t Size>
class RingBuffer {
        double data[Size+1];
        …
        static std::size_t wrap(std::size_t idx) {
                return Size+1;
        }
        …
        std::size_t size() const {
                return (iput >= iget)
                        ? iput - iget
                        : 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<100> b;
RingBuffer<30> b2;
```

```
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


Execution Path taken for Exception
run-time startup
function main
maybe more calls
function catching Ex1
maybe more calls
function possibly throwing Ex1
global initialisation
global cleanup
f1(…)
…
try {
fx(…);
}
catch (Ex1 &e) {
}
local cleanup
if (…)
throw Ex1(…)
Exception Class Hierarchies
Standard C++ exception classes
std::exception
std::runtime_error
only as an example
Ex
Execption class extensions specific to an application or library
Ex1
Ex2
Ex3
try {
}
catch (Ex1 &e) {
}
catch (Ex2 &e) {
}
if (…)
throw Ex1(…);
if (…)
throw Ex2(…);
if (…)
throw Ex3(…);
Exception Classes Viewn as Labels
try {
}
catch (Ex2 &e) {
}
catch (Ex &e) {
}
Grouping Exceptions
try {
}
catch (Ex1 &e) {
}
Enabling Handler Blocks
try {
}
catch (Ex1 &e) {
}
throw;
partial recovery only
Re-throw exception
Catch Any Exception
try {
}
catch (...) {
}
if (…)
throw 42;
if (…)
throw Ex1(…);
if (…)
throw std::runtime_error(…);
Exception Basics

Code-Units (as Stored in Memory)

Historically:

- **narrow (8 bit)** or
- wide (16 bit) characters or
- switching character sets or
- variable length encodings

UTF-8

- *code units are 8 bit wide*
- **7 bit ASCII requires a single (8 bit) byte only**
- characters used in most western languages can be represented in two bytes
- characters from most languages still in use do not require more than three bytes (24 bit)
- no code point uses more than four bytes (32 bit)

UTF-16

- *code units are 16 bit wide*
- characters from most languages still in use are represented in one Code-Unit (16 bits)
- no code point uses more than two code units (32 bit)
- **since UCS2 was dropped in favour of UCS4 the mapping between code points and code units is <u>not</u> any more a 1:1**

UTF-32

- *code units are 32 bit wide*
- **mapping is always 1:1**

(as UCS4 uses 21 bits only, an application might store other character specific data in the remaining 11 bits)

As the mappings are standardized there are library solutions (now also in C++11)

most technical solutions aimed for a 1:1 mapping of code points and code units

Unicode separates the mapping issue (and also defines several standard ways)

Code-Points (as defined for the Character Set)

classic (7-Bit) ASCII

…	59	5A	5B	5C	5D	5E	5F	60	…
…	Y	Z	[	\	]	^	_	`	…

ISO 646-DE ("German" 7-bit ASCII Variant)

…	59	5A	5B	5C	5D	5E	5F	60	…
…	Y	Z	Ä	Ö	Ü	^	_	`	…

ISO 8859-1 (8 bit)

…	5A	5B	5C	…	A4	…	DB	DC	DD	…
…	[	\	]	…	¤	…	Û	Ü	Ý	…

ISO 8859-15 (8 bit)

…	5A	5B	5C	…	A4	…	DB	DC	DD	…
…	[	\	]	…	€	…	Û	Ü	Ý	…

Combining vs. Precomposed Characters (value examples from Unicode)

…	44	…	55	…	5D	5C	…	65	…	308	…	20AC	…	20E5	…
…	C	…	U	…	\	]	…	e	…	̈	…	€	…	\	…

Initial Unicode Specification (16 bits)

UCS2 = 2^{16} = 65536 Code Points

Later Unicode Extension (0x0 … 0x10FFFF with some unused ranges)

UCS4 = $2^{20} + 2^{16} - 2^{11}$
= more than 1 Million Code Points

Various Problems to be solved <u>mainly</u> by Rendering Engine and Input Methods, and some with additional Libraries like ICU (http://ibm.com/software/globalization/icu/)

Visual Appearance (as perceived by user)

1

]

2

¤

€

no precomposed form

3

e

Ü

4

not unique from code points

1) not all characters available at the same time
2) future needs not anticipated
3) combining characters introduced flexibility …
4) … but in combination with their precomposed variants also introduce ambiguities

!

- C++ character and string types are templated on the code unit size.
- No information about the **character set** is carried in the type!
- Furthermore, type `wchar_t` is implementation defined.

!

- Behaviour of rendering engine and input method configured externally.
- File I/O-conversions may apply globally.

Code-Units vs. Code-Points

Standard Strings 101

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

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 explicitely converted ...

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

… the other way round is automatically

Classes

CharType
std::basic_string

Lookup in reference documentation here …

char
std::basic_string

`std::string`

… but prefer these `typedef`-s for readability!

wchar_t
std::basic_string

`std::wstring`

Efficiency

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

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

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.

May or May Not Have COW-Implementation

a → `hi!`
b → `Hello, world`

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

`a = b;`

becomes (silently) ***shared***

a, b → `Hello, world`

content copied on write ...

`a[7] = 'W';`

… and again ***unshared***

a → `Hello, World`

Input and Output

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

For output
- `operator<<` has the usual behaviour.

```
using namespace std;
…
// read standard input by line
string line;
while (getline(cin, line))
    …
```

```
// to given separator
string w;
istringstream s(line);
… getline(s, w, ':') …
```

```
// by word
… cin >> w …
```

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

Numeric Conversions

C++11

```
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) …
…
```

convert to every builtin **integral** type

convert to every builtint **floating point** type

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

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

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;
- hoth 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`, …

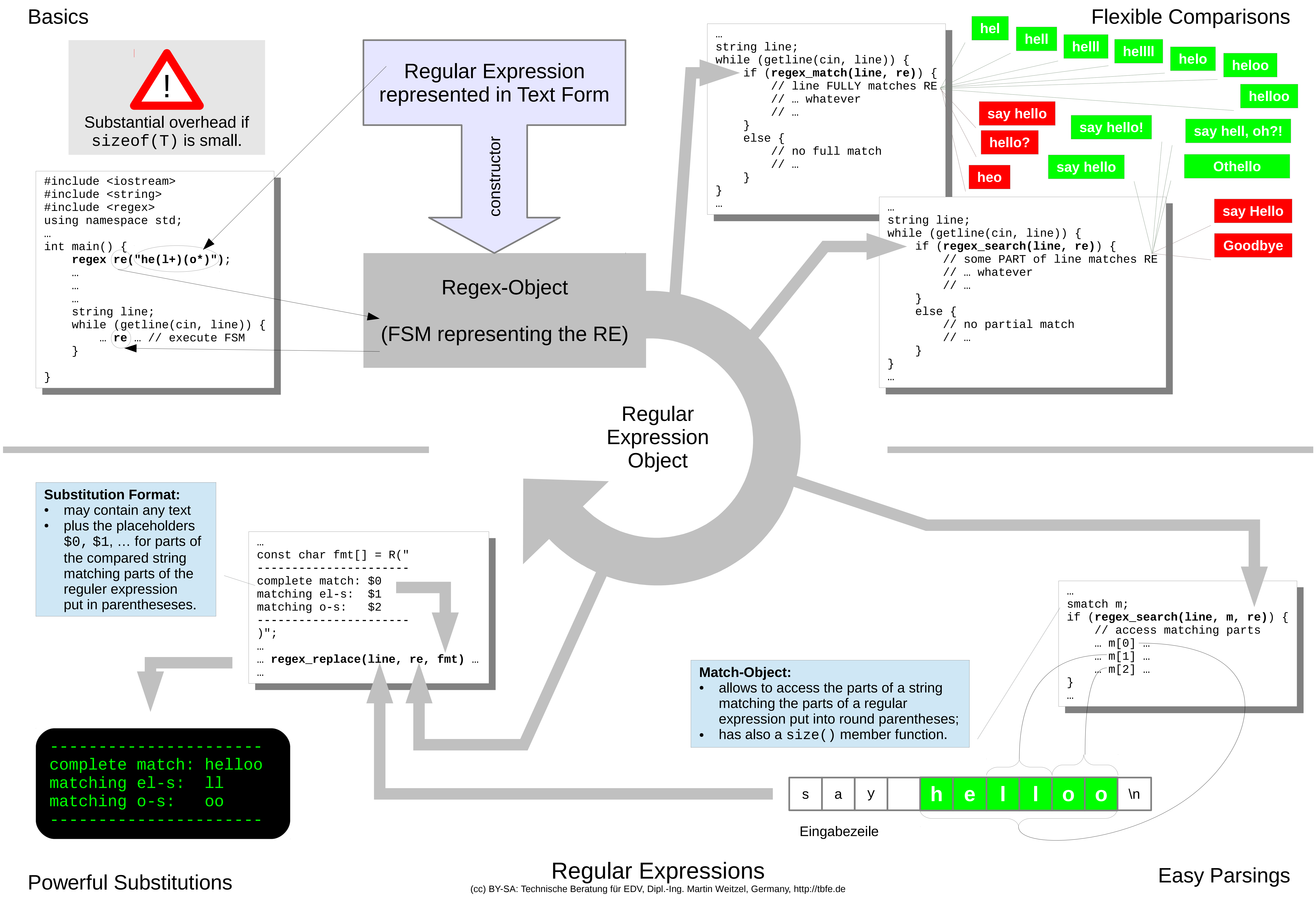
```
std::string str;
…
for (char c : str) {
    // process str
    // char-by-char
    …
}
```

Advanced String Operations

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.

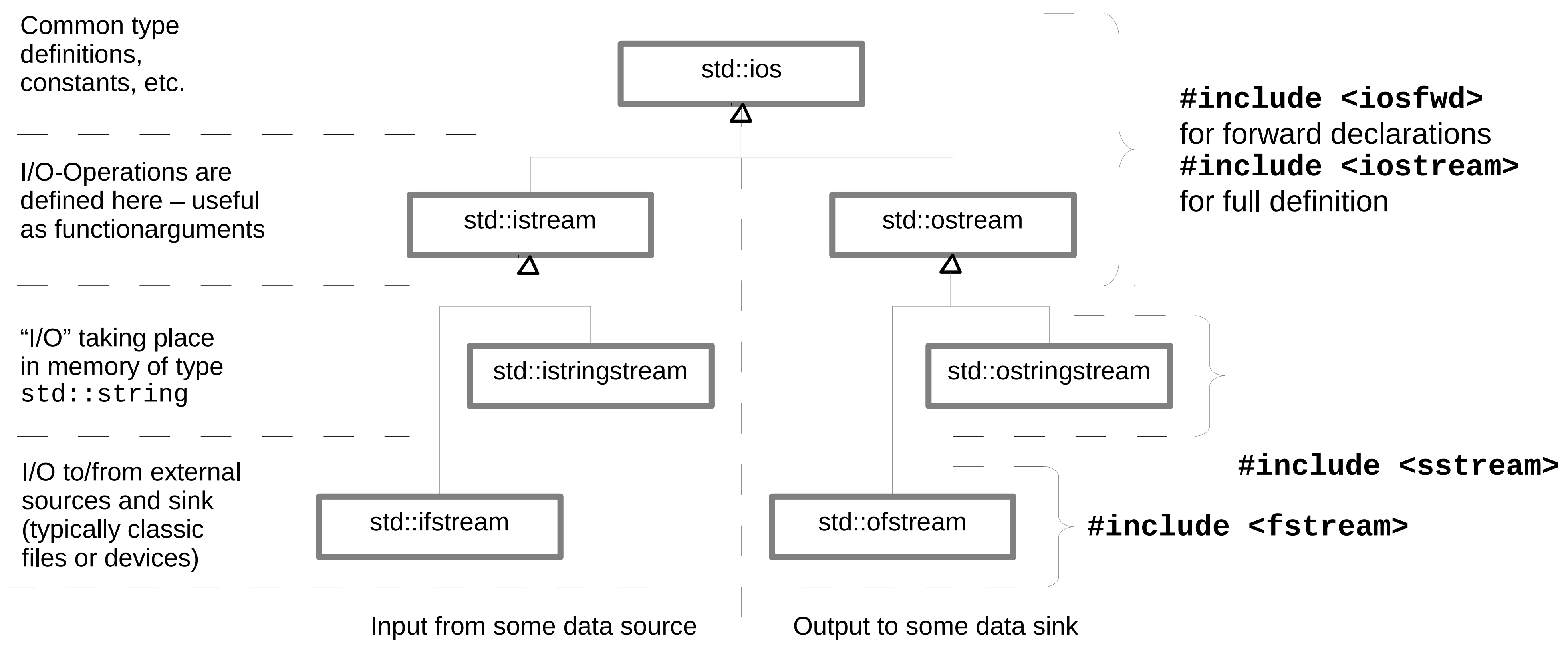
C++11 has added some "missing" ones:
- e.g. get rid of over-allocation with `shrink_to_fit`



Regular Expressions

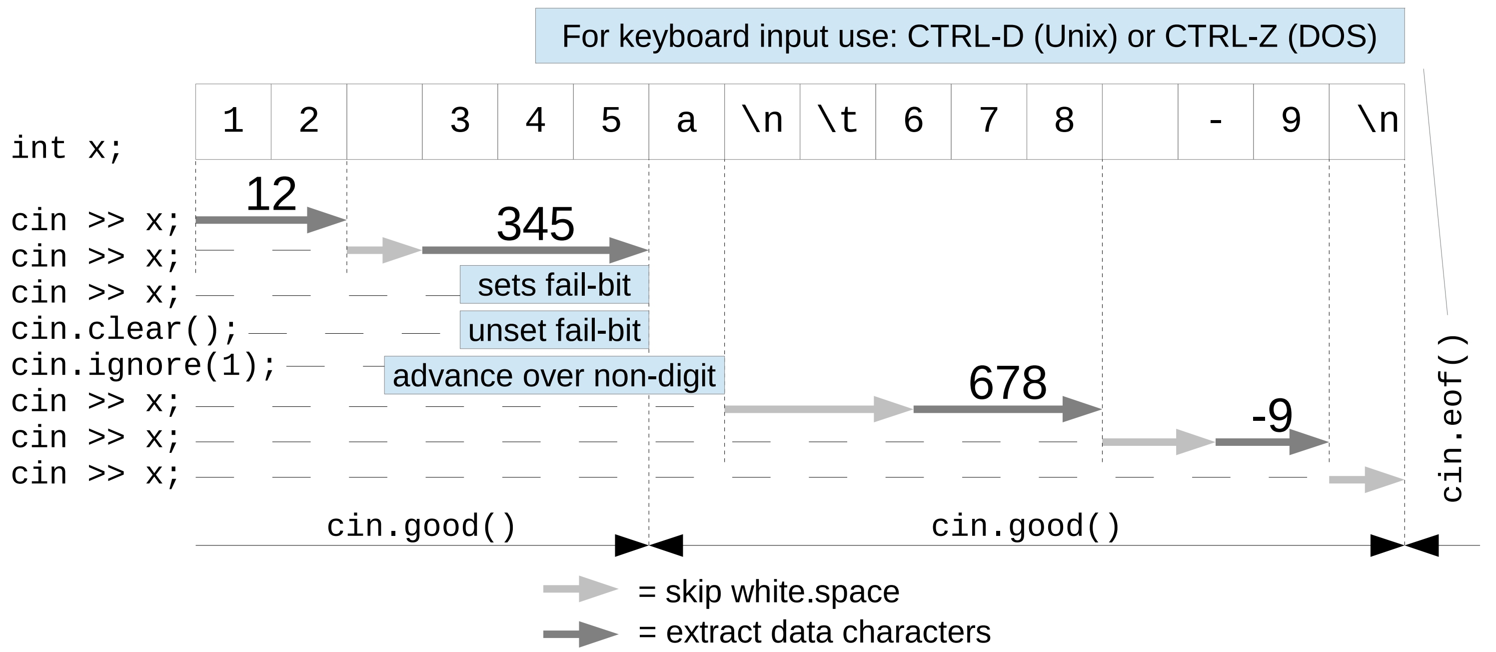
I/O-Stream Basics

I/O-Streams Front-End



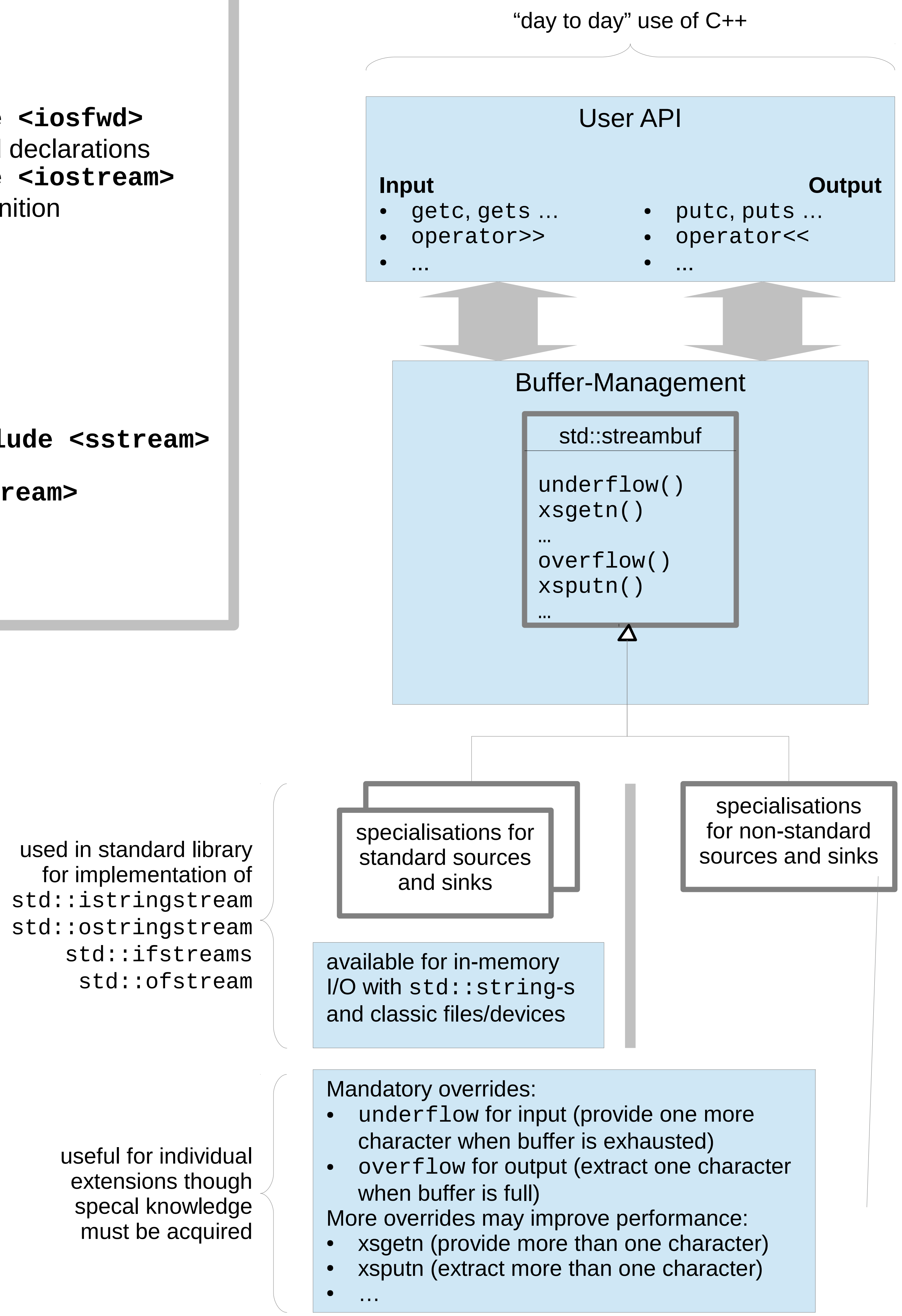
I/O-Stream States (assuming namespace `std` and stream named *s*)

Set ...	Name	is set ?	set explicitely	all unset ?	unset all
… on end of input	`ios::failbit`	*s*`.fail()`	*s*`.clear(ios::failbit)`		
… on format error	`ios::eofbit`	*s*`.eof()`	*s*`.clear(ios::eofbit)`	*s*`.good()`	*s*`.clear()`
(implem. defined)	`ios::badbit`	*s*`.bad()`	*s*`.clear(ios::badbit)`		



I/O-Streams State-Bits

I/O-Streams Back-End

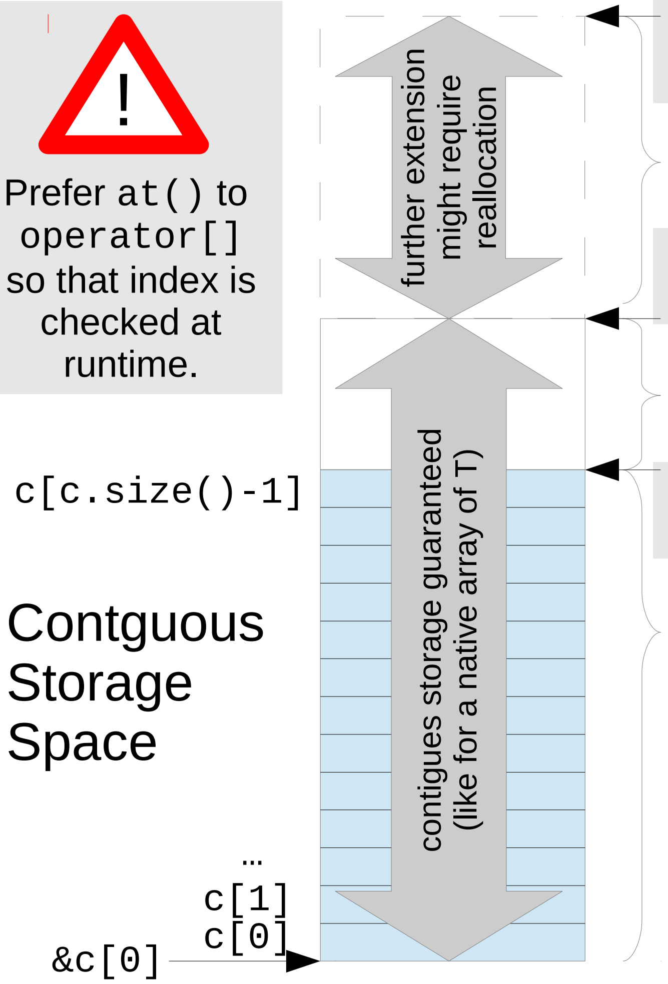


STL – Sequence Container Classes

Contiguous Storage Space

`std::vector<T> c;`

Prefer `at()` to `operator[]` so that index is checked at runtime.



up to here theoretically space for `c.max_size()` elements (physical memory might enforce a lower limit)

more memory potentially available

up to here space for `c.capacity()` elements (change with `c.reserve()`)

pre-allocated but not yet initialized memory space

up to here space for `c.size()` elements (change with `c.resize()`)

`c.size()` objects of type `T` at least initialised with constructor

Typical implementation of data structure:
- three pointers to
  - `&c[0]`
  - `&c[0] + c.size()`
  - `&c[0] + c.capacity()`

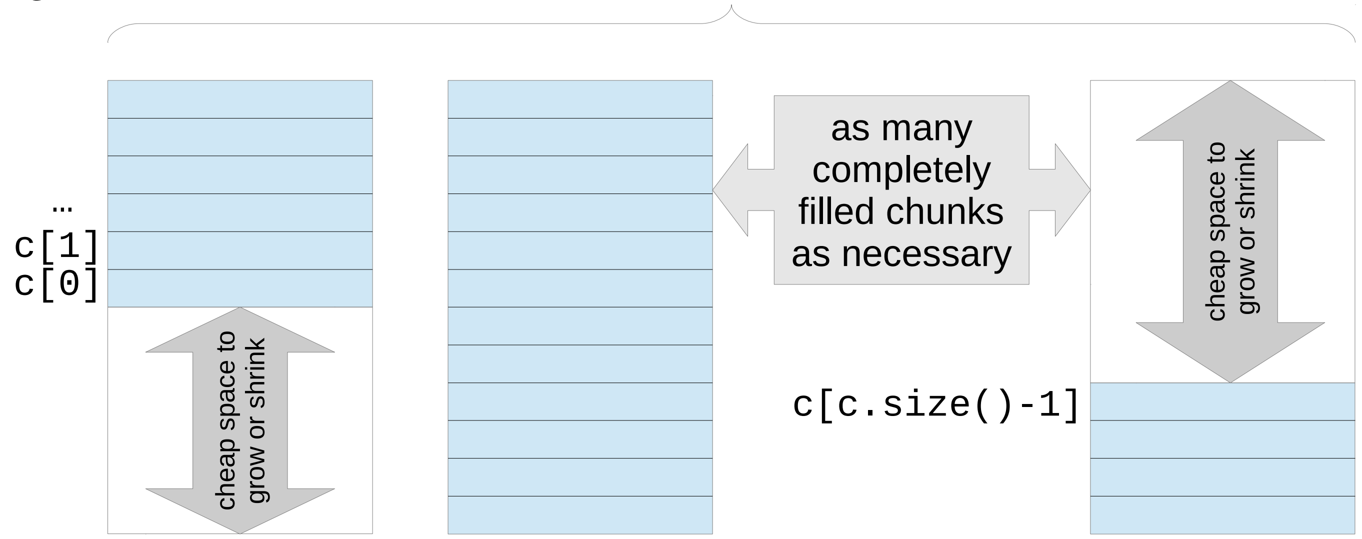
Direct element access (in addition to dereference):
- **pointer arithmetic only**

Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by address arithmetic and assignment

Double-Ended Queue

`std::deque<T> c;`

`c.size()` objects of type `T` at least initialised with constructor and typically some pre-allocated space before first and after last element



Prefer `std::vector` if insertion/removal always takes place at the same end

Typical implementation of data structure:
- pointer to first and last element
- one more pointer to
  - **additional block holding pointers to chunks**
- integral value for number of elements

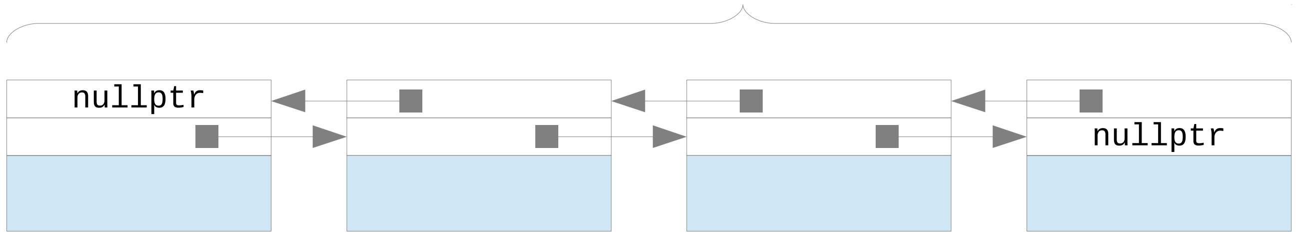
Direct element access (in addition to dereference):
- presumably some “masking and shifting”
- indirect memory access
- address arithmetic

Minimal implementation of iterator: one pointer
Advancing iterators requires memory access and test followed by either address arithmetic or assignment

Double Linked List

`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

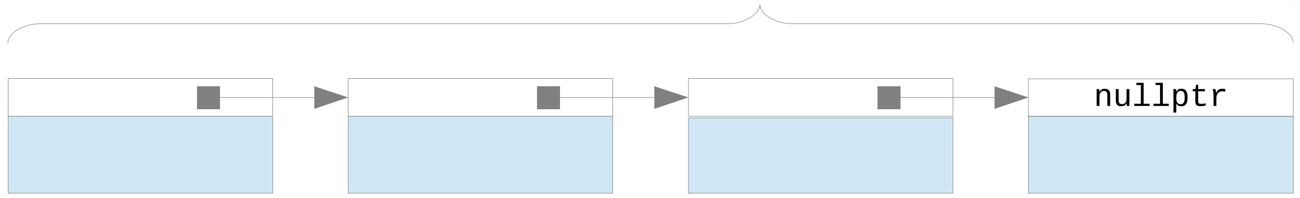
Direct element access not supported!
Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by assignment

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

Singly Linked List

`std::forward_list<T> c;`

objects of type `T` initialised with constructor

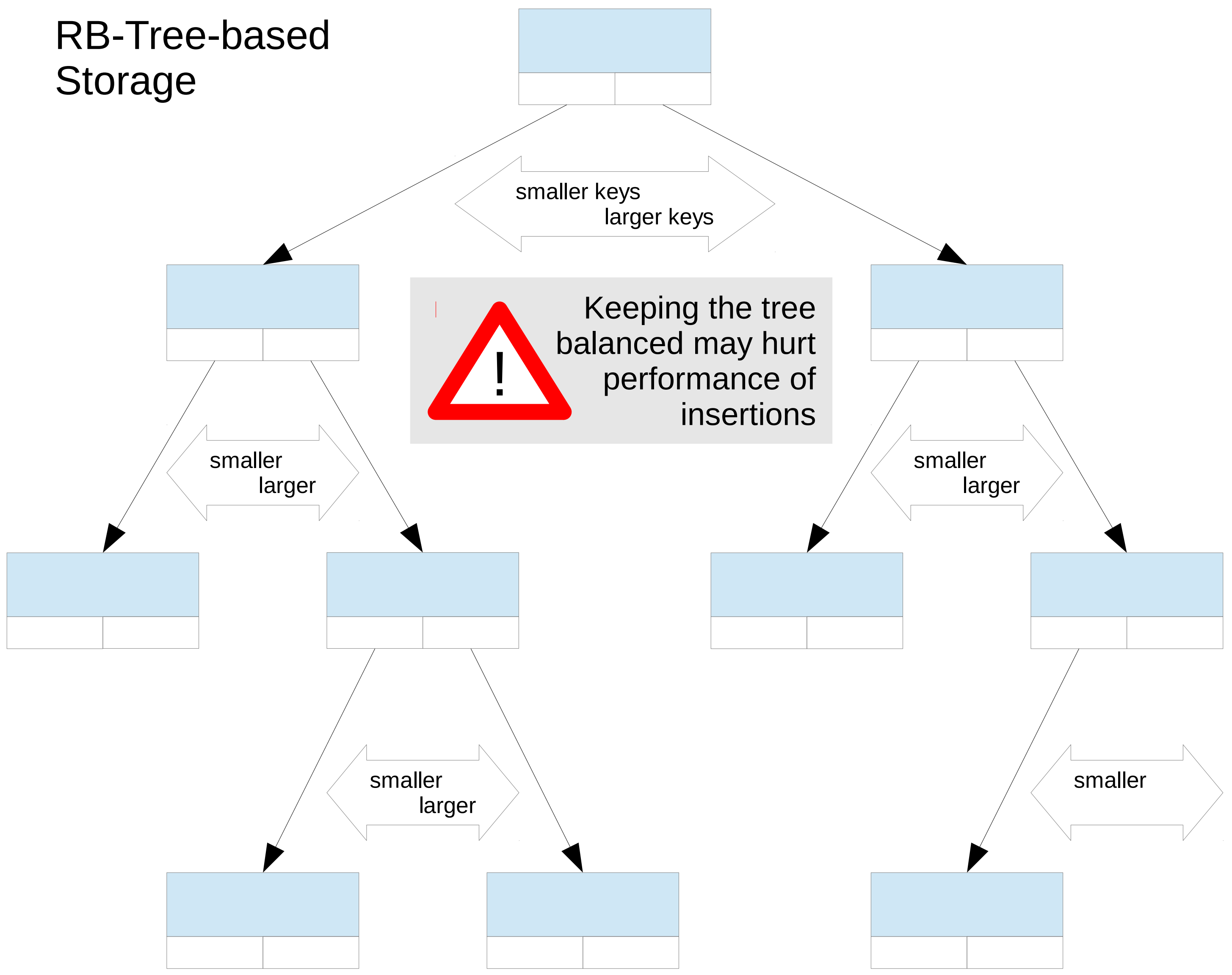


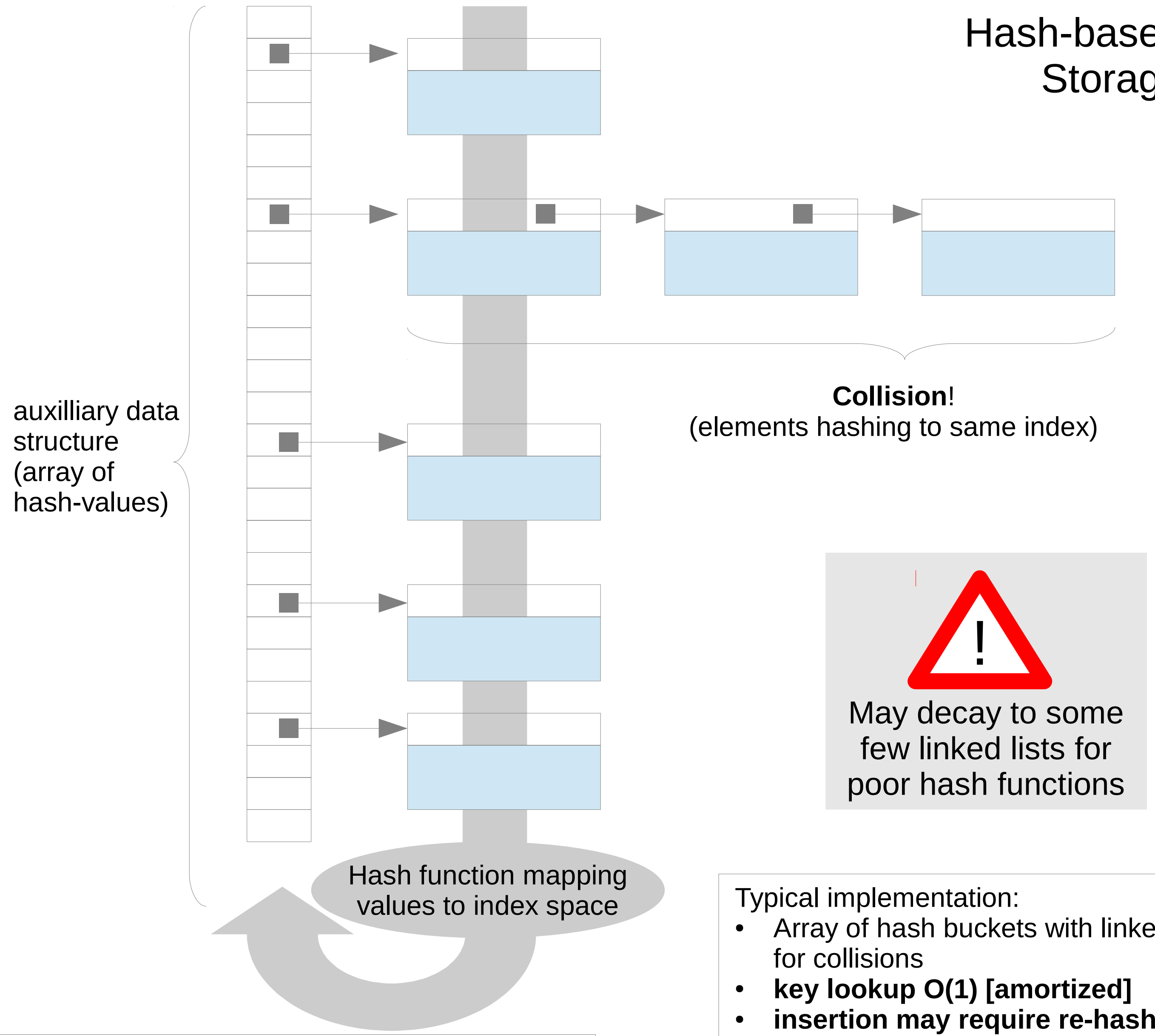
Use `c.empty()` to check wether elements exist.

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

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



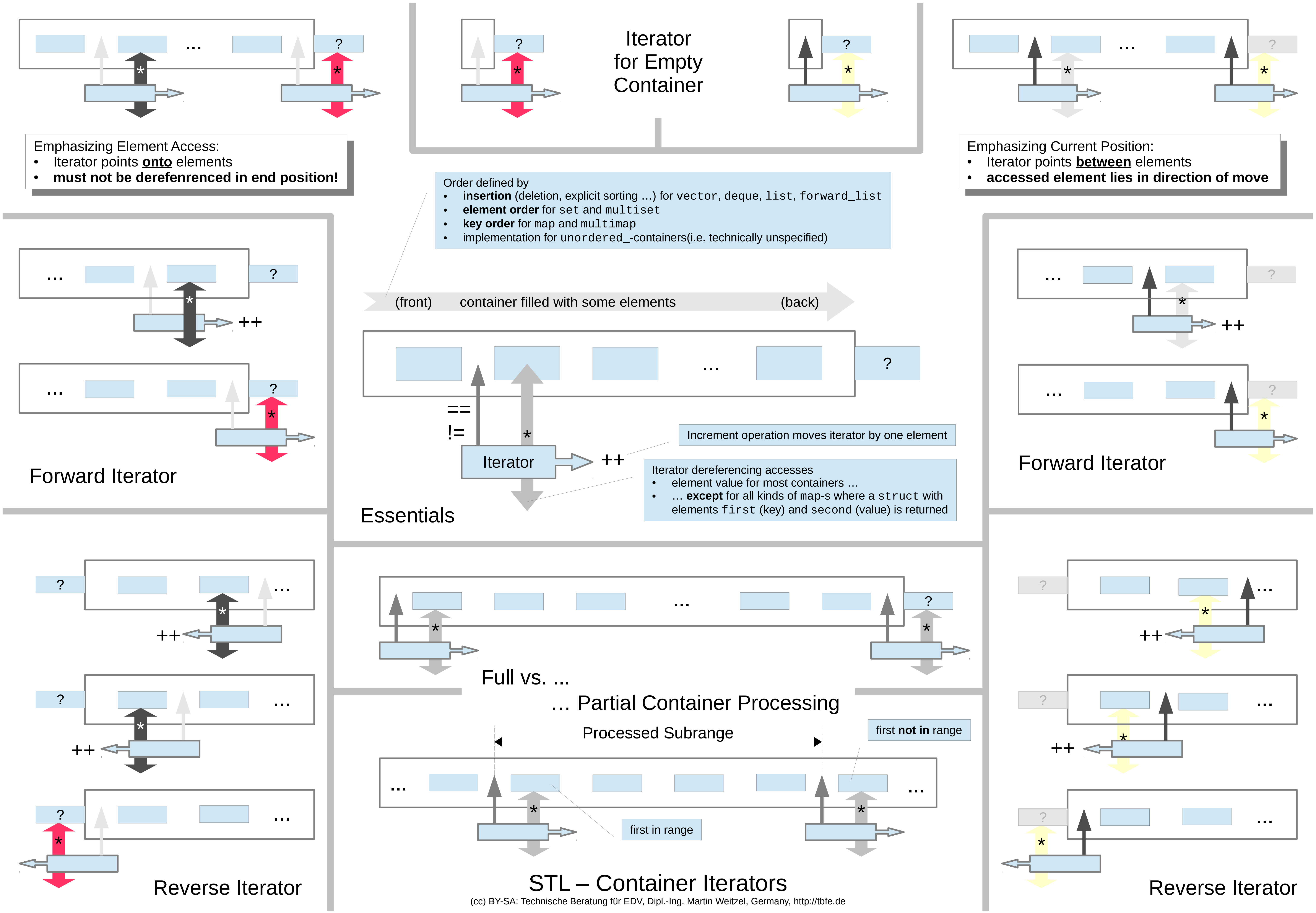


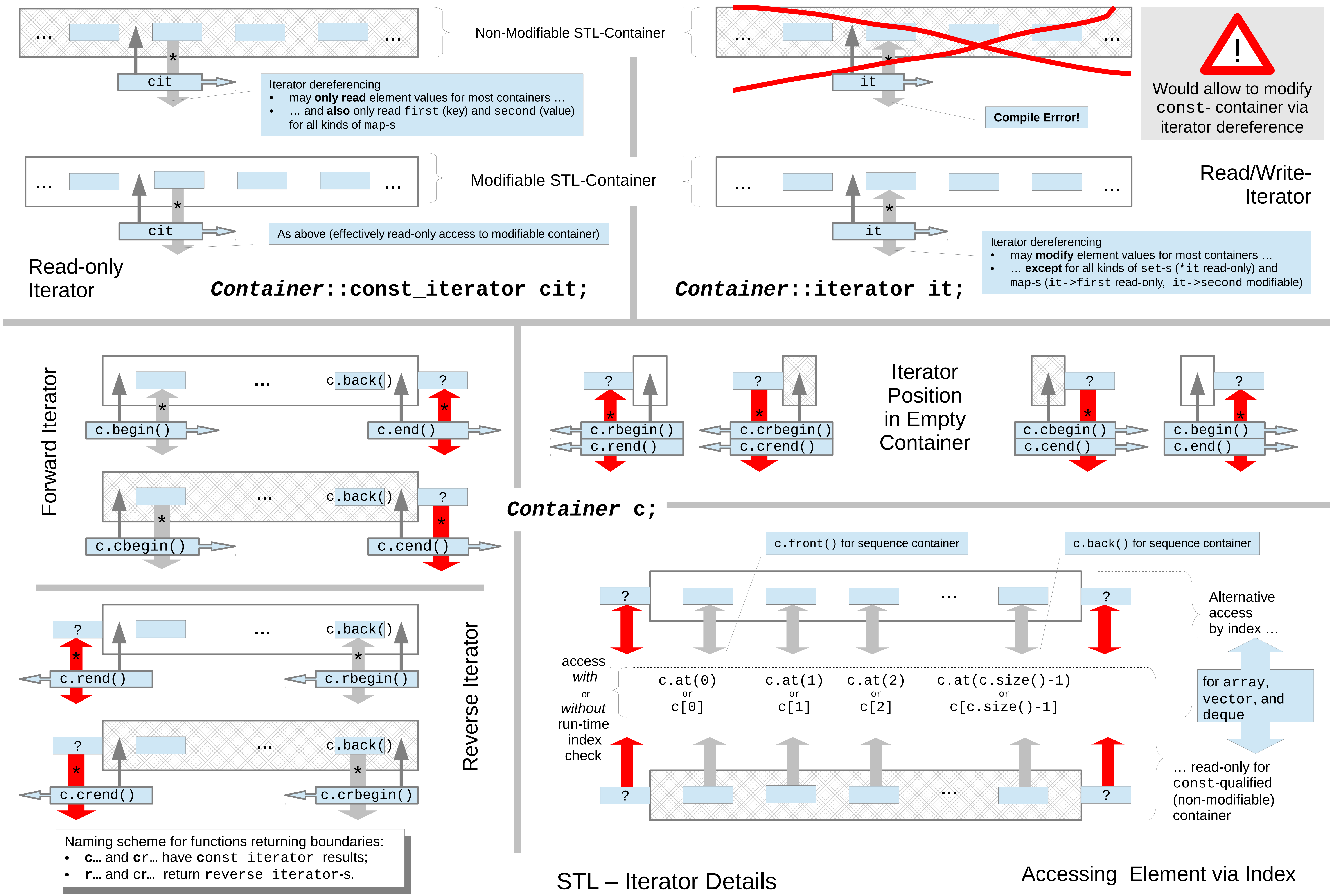
Typical implementation: Black-Red-Tree
- **key lookup O($\log_2$ N)**
- **insertion may require re-balancing**
- two pointers per element

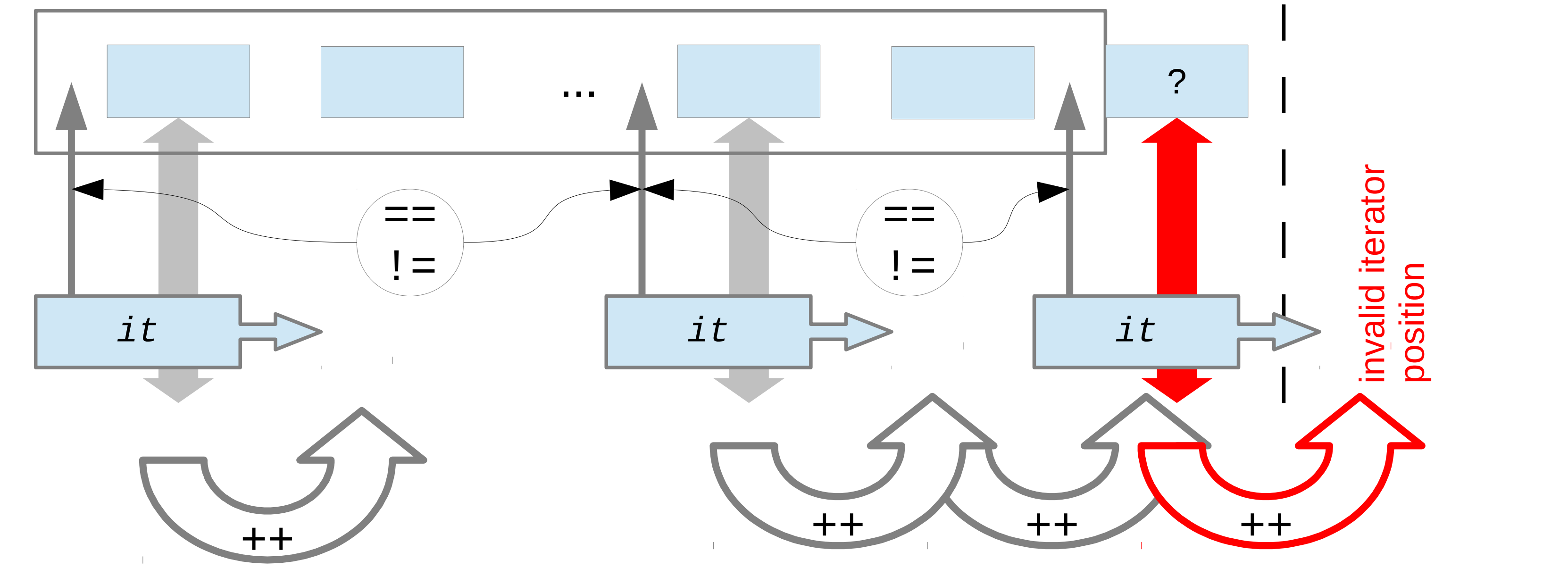
Minimal implementation of iterator: single pointer (but may be more for an effiient implementation). Advancing iterators requires some memory accesses and tests depending on the location of the node in the tree or hash bucket list, followed by assignment.

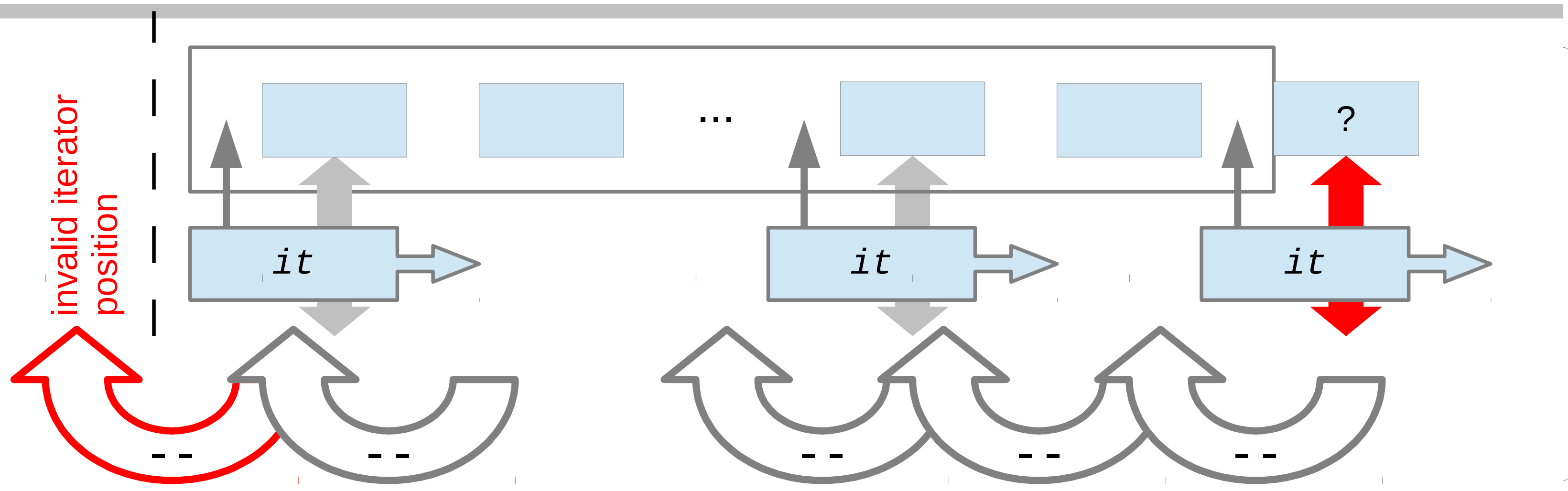
Typical implementation:
- Array of hash buckets with linked lists for collisions
- **key lookup O(1) [amortized]**
- **insertion may require re-hashing**
- one pointer per element
- for good performance ~20% oversized array of pointers for maximum number of elements

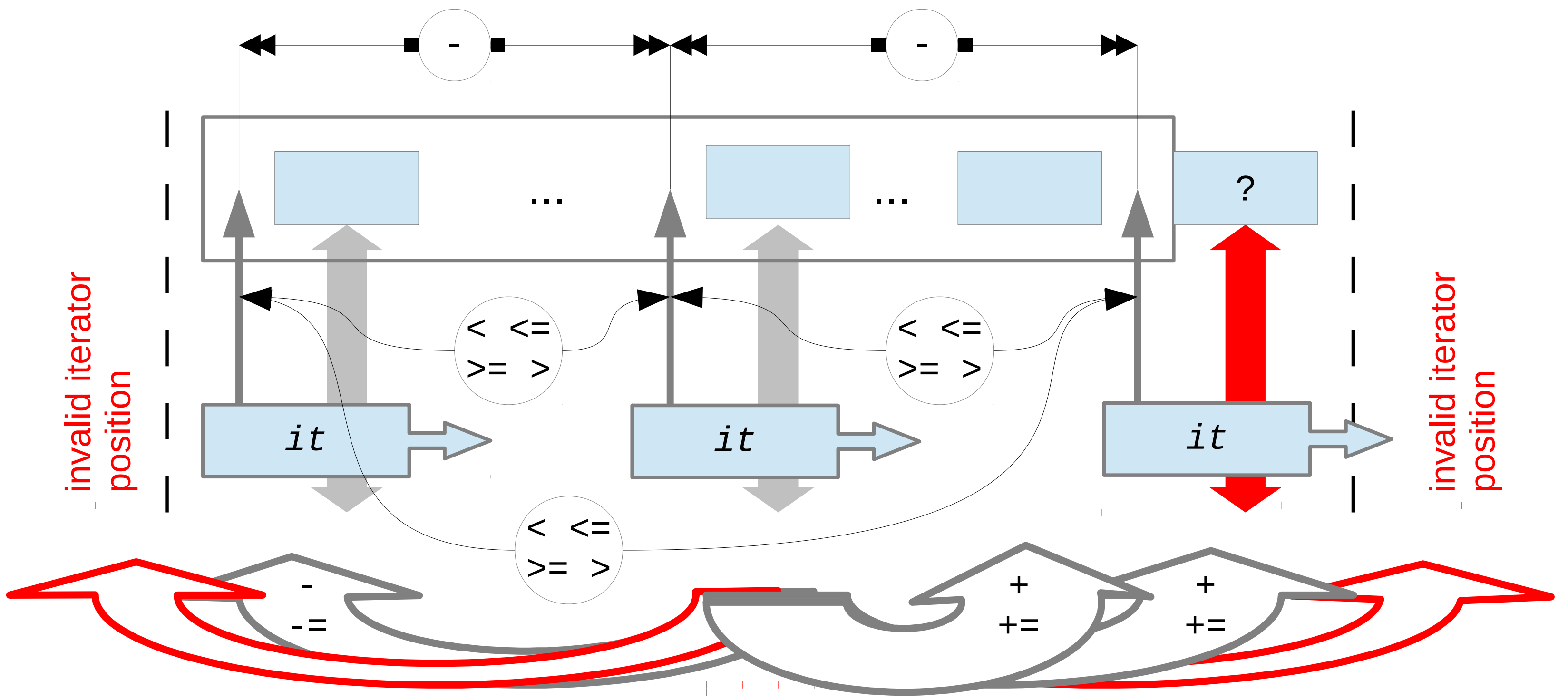
STL – Associative Container Classes


STL – Container Iterators
Emphasizing Element Access:
• Iterator points onto elements
• must not be derefenrenced 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)
Iterator
==
!=
++
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
Full vs. …
… Partial Container Processing
Processed Subrange
first not in range
first in range
Reverse Iterator
Reverse Iterator


Non-Modifiable STL-Container
cit
Iterator dereferencing
• may only read element values for most containers …
• … and also only read first (key) and second (value) for all kinds of map-s
it
Compile Errror!
Would allow to modify const- container via iterator dereference
Modifiable STL-Container
cit
As above (effectively read-only access to modifiable container)
Read-only Iterator
Container::const_iterator cit;
it
Read/Write-Iterator
Iterator dereferencing
• may modify element values for most containers …
• … except for all kinds of set-s (*it read-only) and map-s (it->first read-only, it->second modifiable)
Container::iterator it;
Forward Iterator
c.back()
c.begin()
c.end()
c.cbegin()
c.cend()
Iterator Position in Empty Container
c.rbegin()
c.rend()
c.crbegin()
c.crend()
c.cbegin()
c.cend()
c.begin()
c.end()
Container c;
Reverse Iterator
c.rend()
c.rbegin()
c.crend()
c.crbegin()
Naming scheme for functions returning boundaries:
• c… and cr… have const iterator results;
• r… and cr… return reverse_iterator-s.
c.front() for sequence container
c.back() for sequence container
access with or without run-time index check
c.at(0) or c[0]
c.at(1) or c[1]
c.at(2) or c[2]
c.at(c.size()-1) or c[c.size()-1]
Alternative access by index …
for array, vector, and deque
… read-only for const-qualified (non-modifiable) container
STL – Iterator Details
Accessing Element via Index







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	

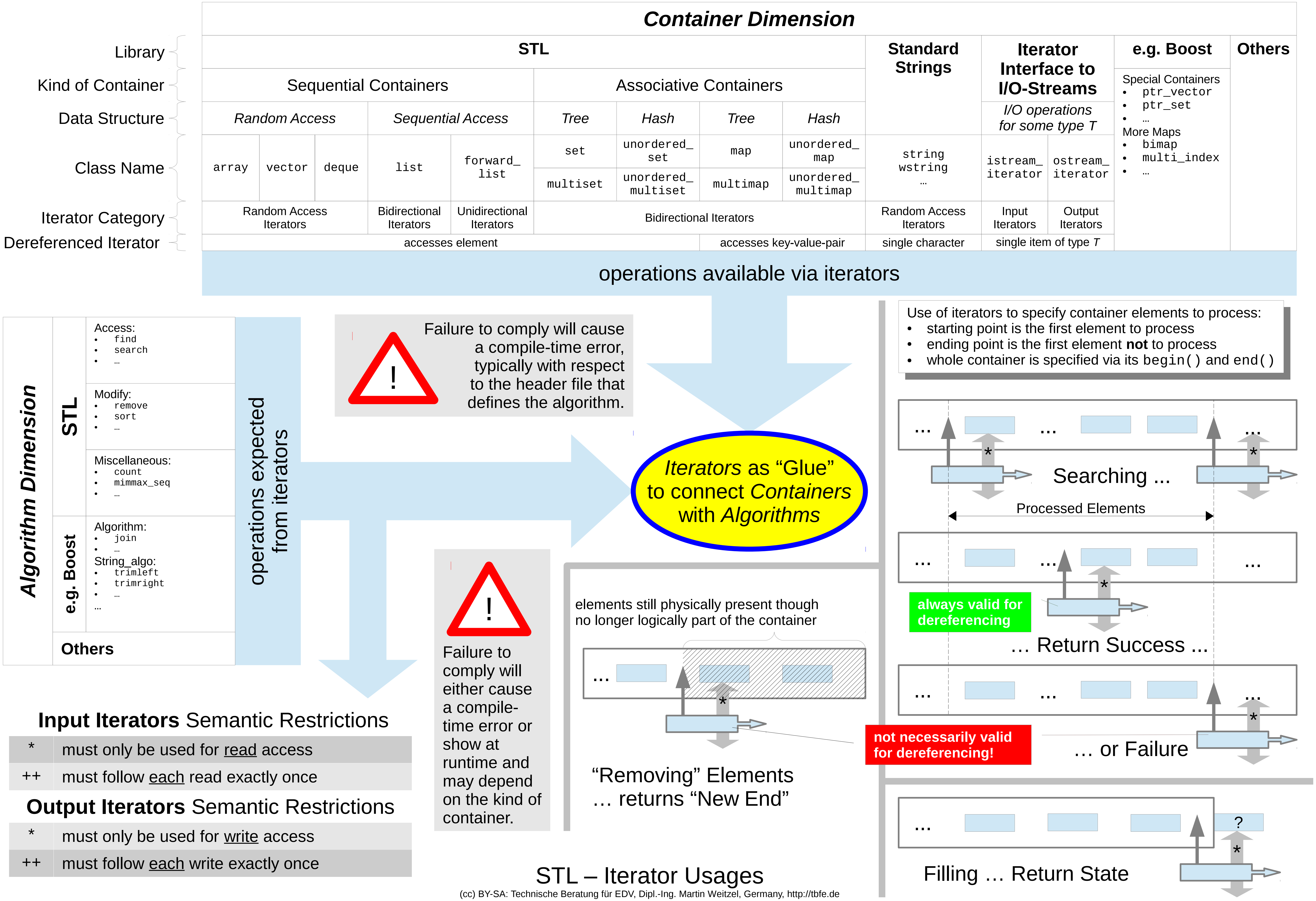
Additional Operations of **Bidirectional Iterators**

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

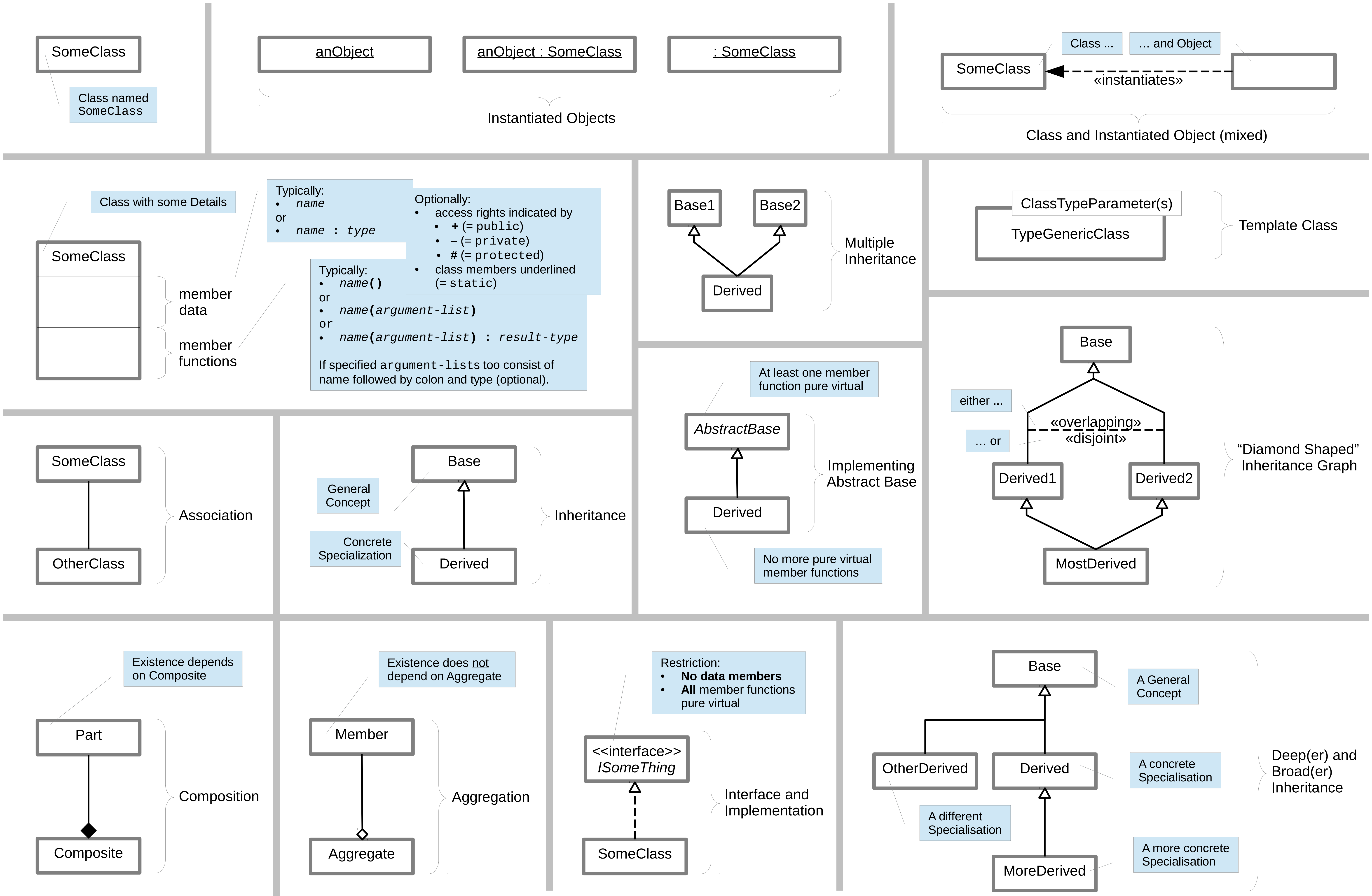
Additional Operations of **Random Access Iterators**

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

STL – Iterator Categories



STL – Iterator Usages



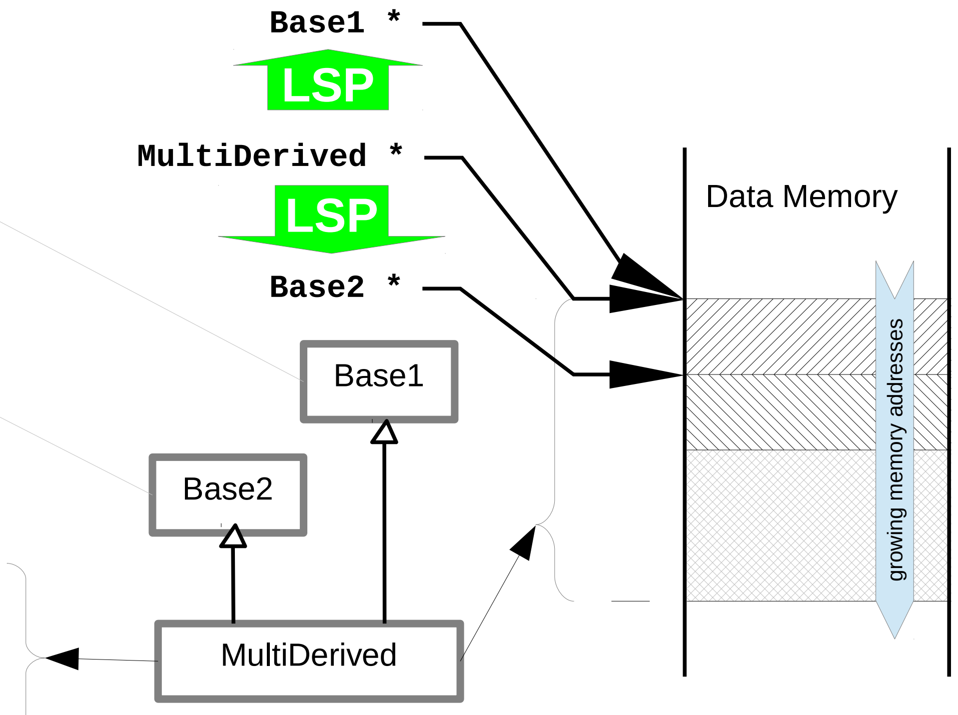
UML – Classes and Relations

Multiple Inheritance

```
class Base1 {
};
```

```
class Base2 {
};
```

```
class MultiDerived
    : public Base1,
      public Base2 {
};
```

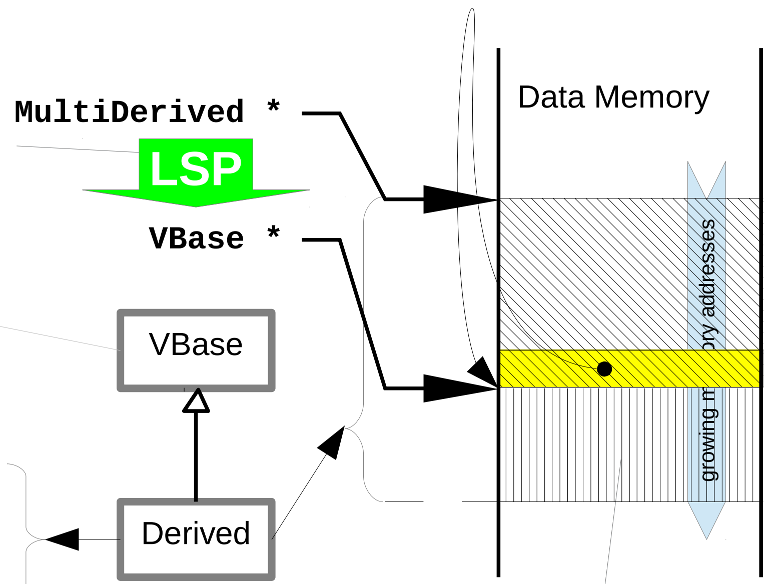


Virtual Base Class

No LSP for `virtual` **`private`** base classes.

```
class VBase {
};
```

```
class Derived
    : virtual public VBase {
};
```



Memory layout is the same for `virtual` **`private`** base classes.

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.

Virtual base classes and resulting memory layout only shows one of several possible implementations

Overlapping Common Base Class

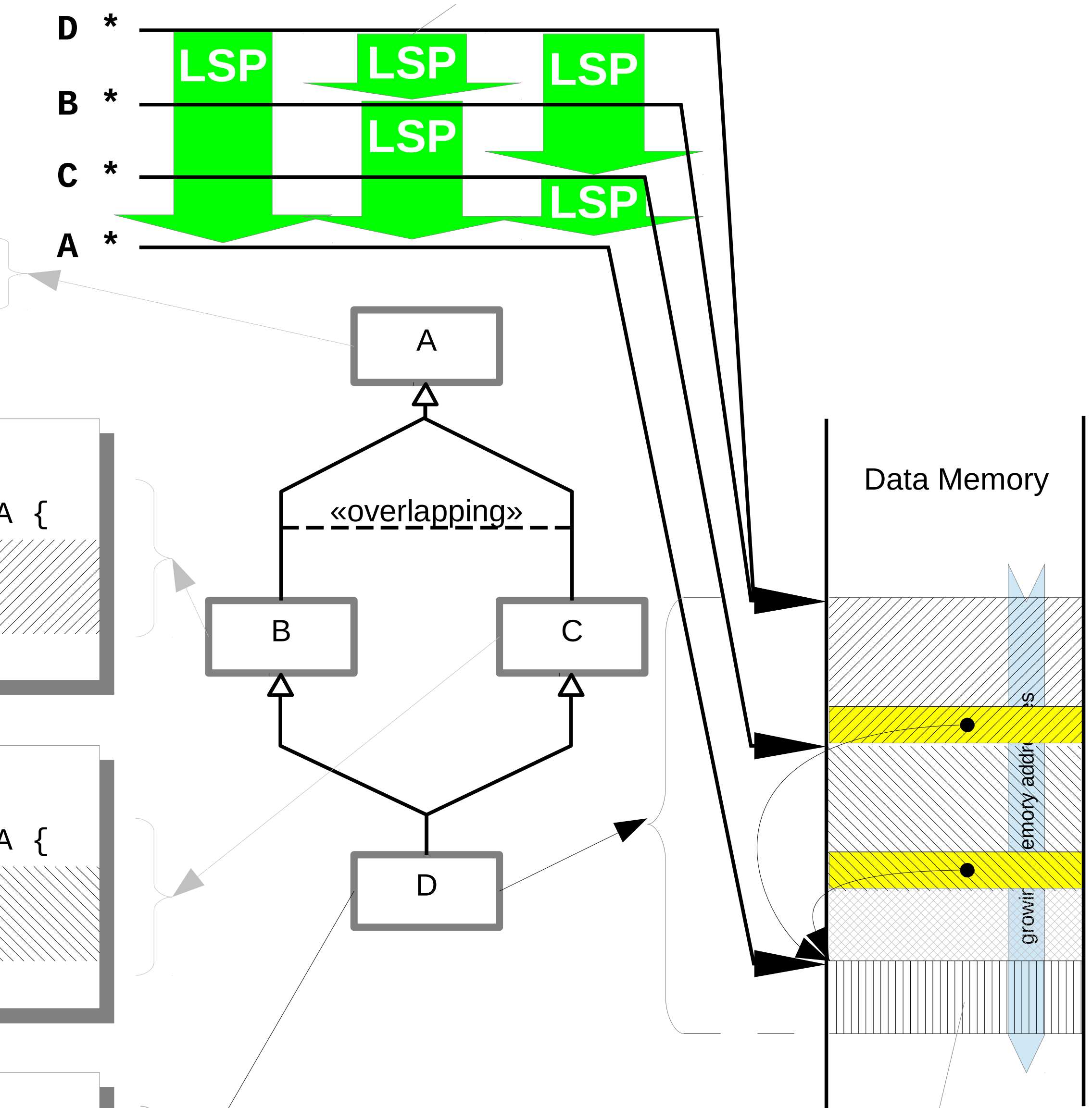
No LSP for `virtual` **`private`** base classes.

```
class A {
};
```

```
class B
    : virtual
      public A {
};
```

```
class C
    : virtual
      public A {
};
```

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

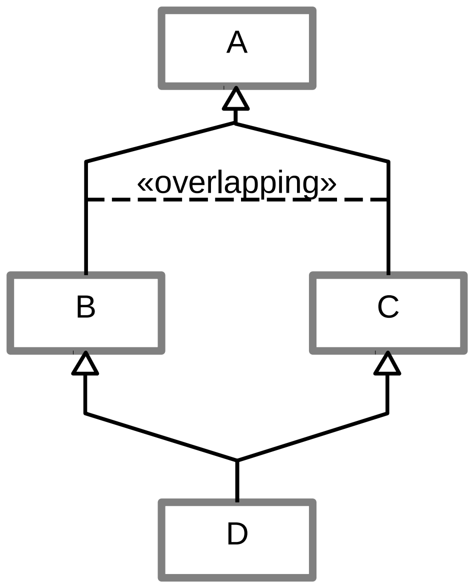


Memory layout is the same for `virtual` **`private`** base classes.

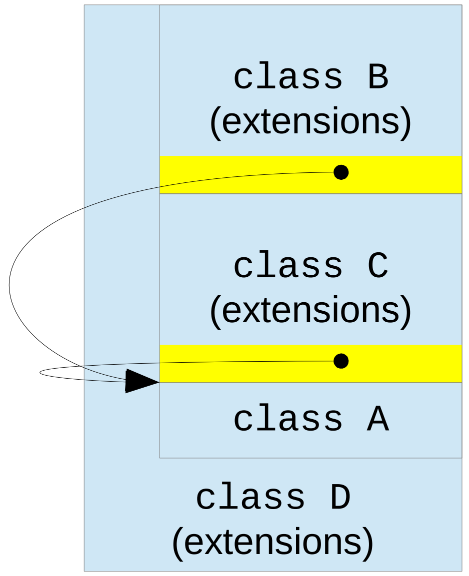
Virtual base classes are the mechanism to make a common base in a "diamond-shaped" inheritance relationship overlapping (see `A` above).

- 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 offets.
- 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

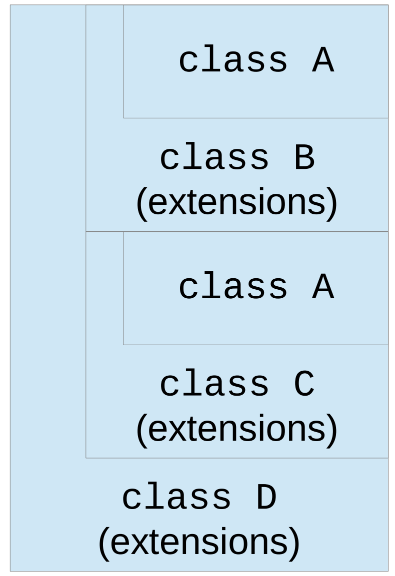


Member Data to Memory Mapping
(showing *one of several possible* solutions)

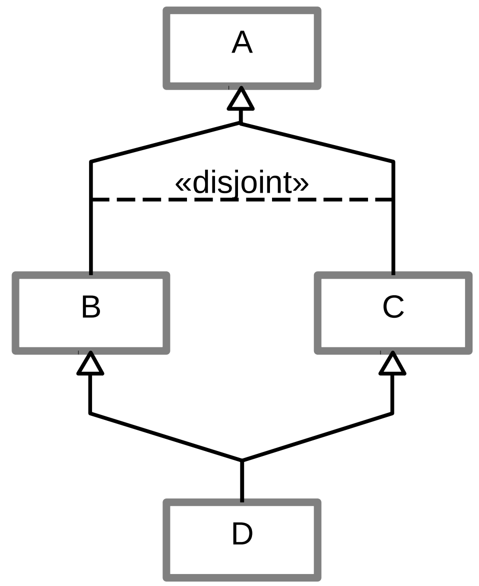
Automatic Type Conversions

to ←	*from*	→ *to*
A	A	A
A	B	A
A	C	A
A, B, C	D	B, C

Up-Casts by LSP



Member Data to Memory Mapping
(showing the *straight forward* solution)



UML Class Graph

C++ Source

```
class A {
    …
};
class B : virtual public A {
    …
};
class C : virtual 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( … )
{ … };
```

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.

Creation and Destruction of D objects

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()`	

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.**

C++ Source

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

```
A::A( … ) { … };
```

```
A::A( … ) { … };
```

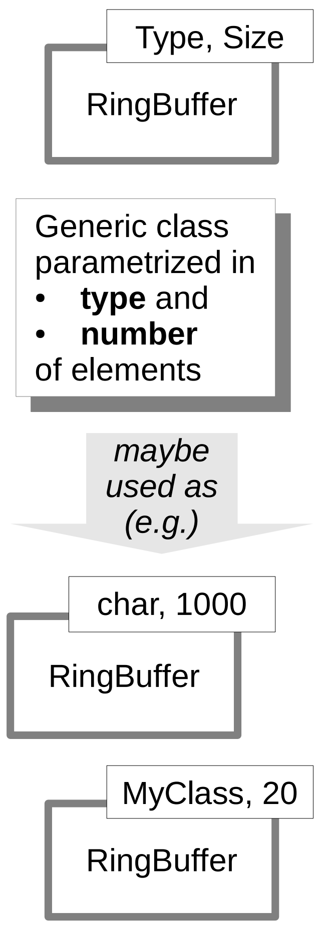
```
B::B( … )
    : A( … )
{ … };
```

```
C::C( … )
    : A( … )
{ … };
```

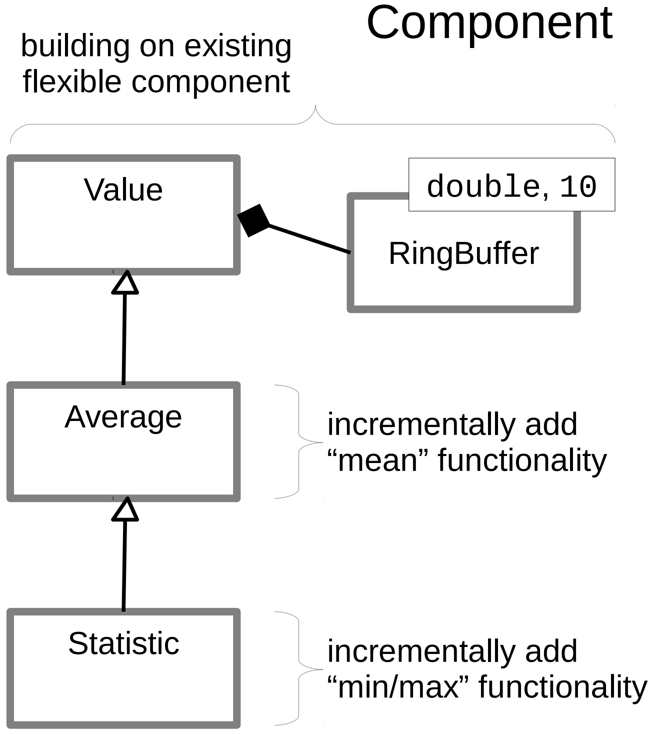
```
D::D( … )
    : B( … ), C( … )
{ … };
```

Diamond Shaped Inheritance

Reusable Component

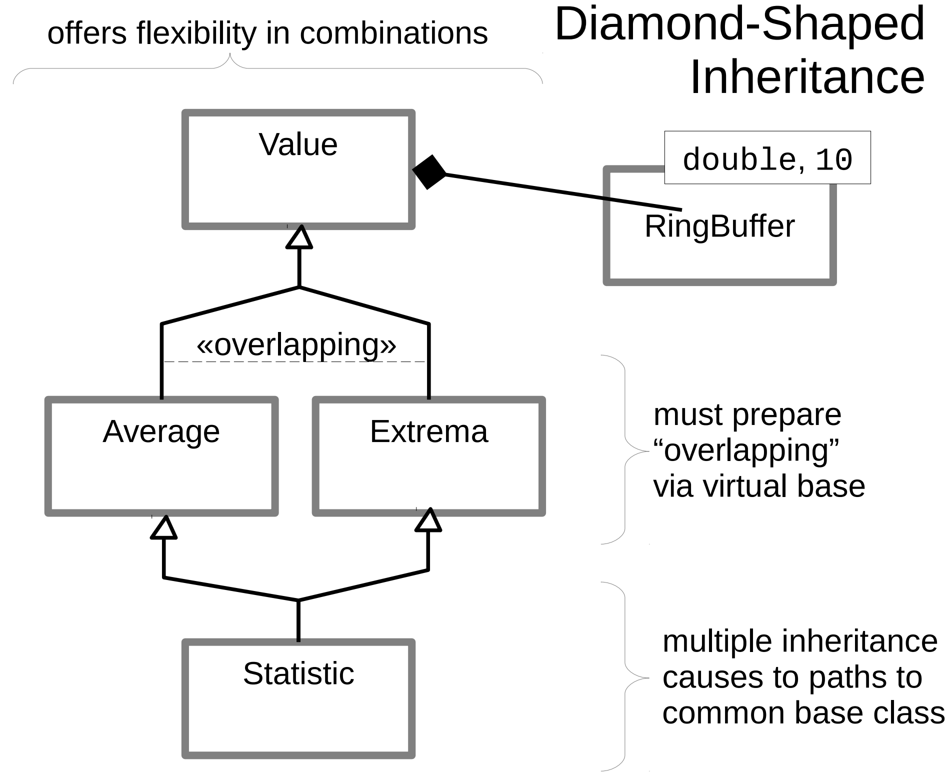


Reusing Adapted Component



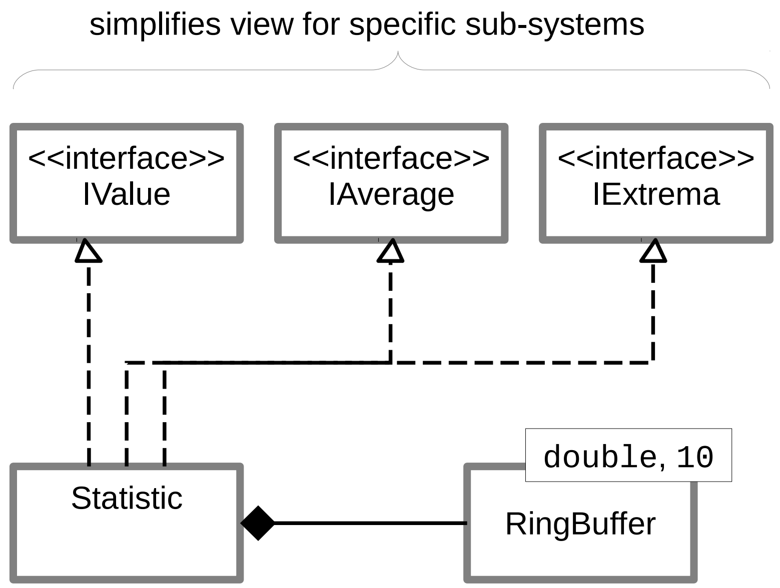
Simple design using:
- *Template Instantiation* (of `RingBuffer`)
- *Composition* (Value <u>has a</u> `RingBuffer`)
- *Inheritance* (`Average` <u>is a</u> `Value` and `Statistic` <u>is an</u> `Average`)

Diamond-Shaped Inheritance



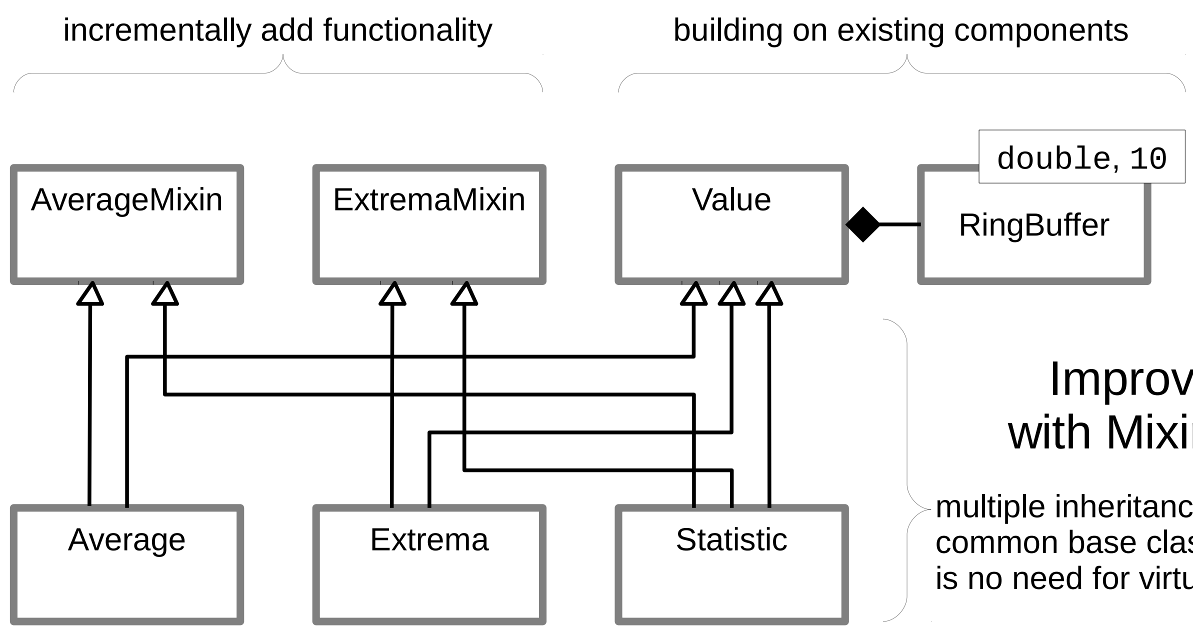
More flexible design with "diamond shaped" inheritance:
- each of the classes (*Value*, *Average*, *Extrema*, *Statistic*) may be used on its own
- intermediate classes (*Average*, *Value*) must pay the "price" …
- … for simple re-use in the most derived class (*Statistic*)

Three Interfaces



Alternative design with interfaces reducing coupling to clients, that do not need to know all the details:
- some clients may only need to handle Values (→ to know `IValue` is sufficient)
- others may need to handle Averages (→ to know `IAverage` is sufficient)
- YetI others may need to handle Extremas (→ to know `IExtrema` is sufficient)

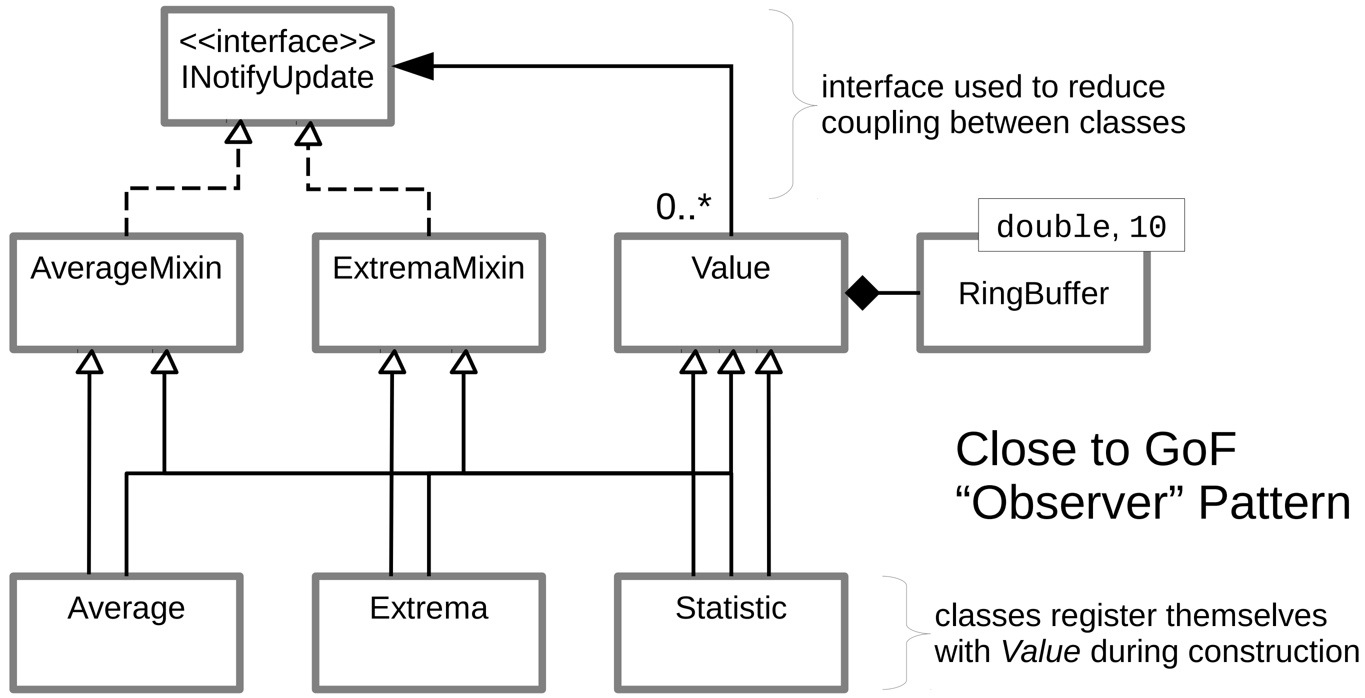
Improved Reuse with Mixin Classes



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

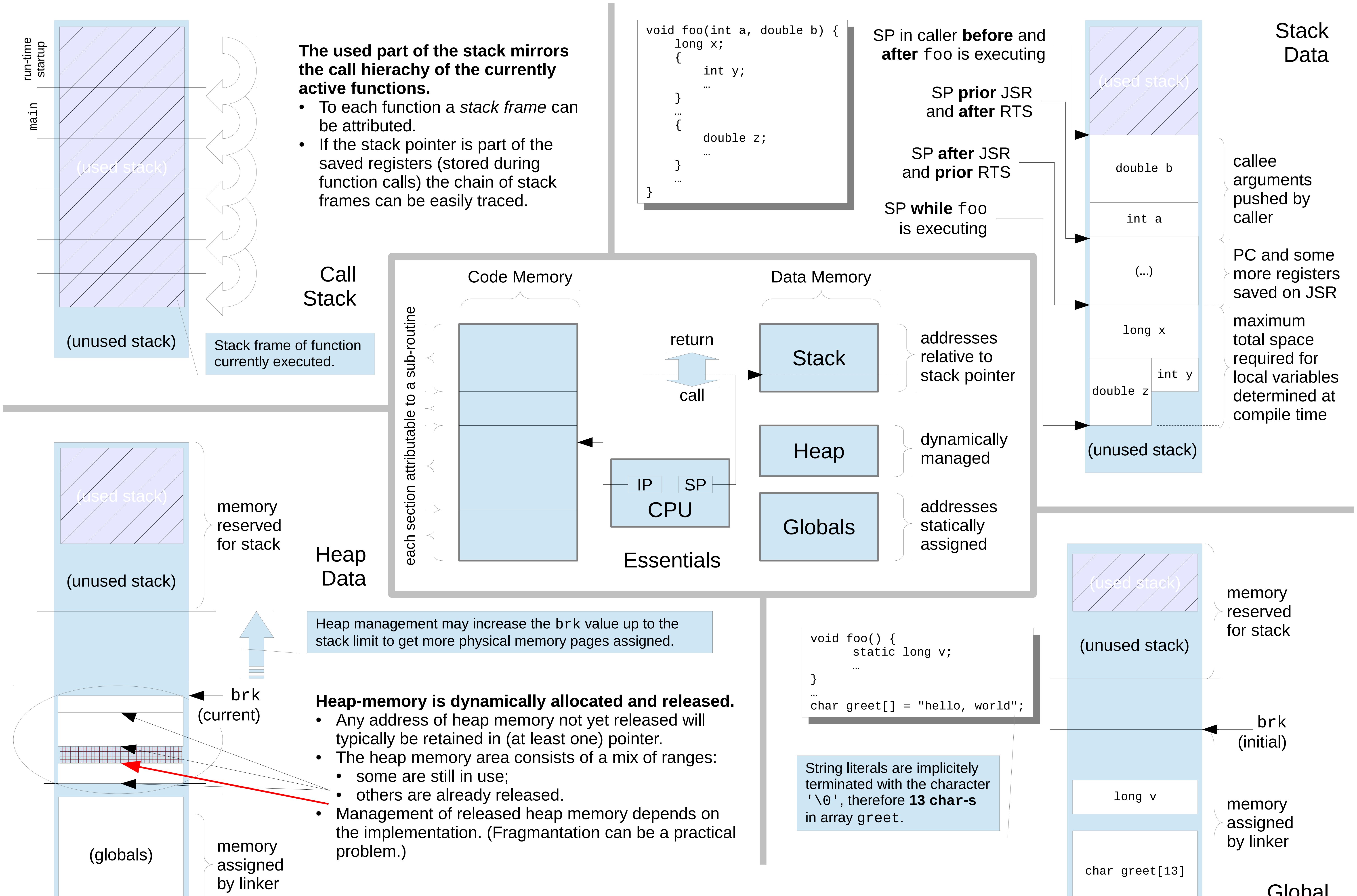
More elaborate design:
- flexibility achieved with "mixin" classes
- multiple inherintance but not "diamand shaped"

Close to GoF "Observer" Pattern



Still more elaborate design:
- Mixins notified via generic interface
- *Value* only handles *INotifyUpdate*

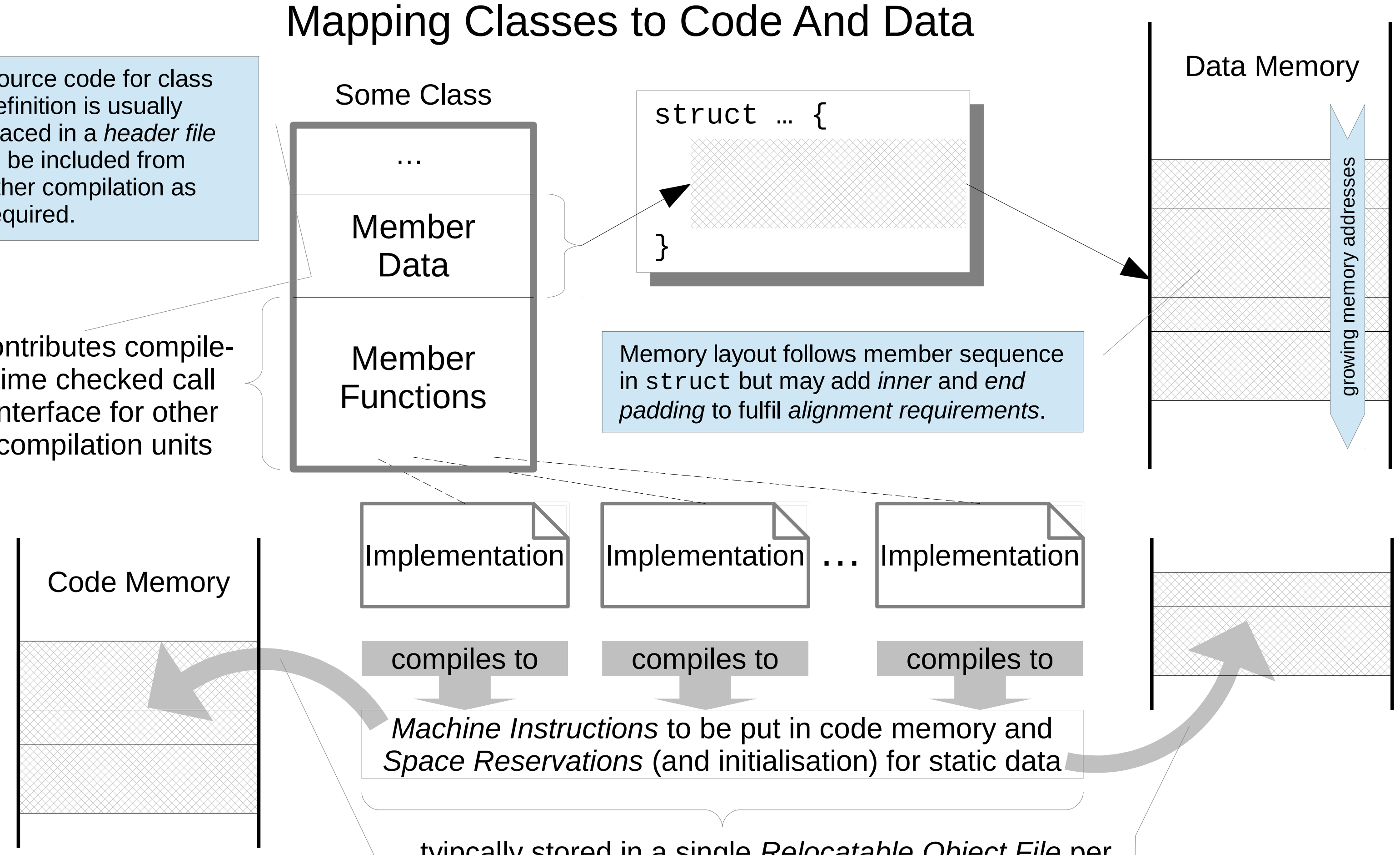
Examples – Classes and Relations



Generalised Execution Model

Mapping Classes to Code And Data

Source code for class definition is usually placed in a *header file* to be included from other compilation as required.



Memory layout follows member sequence in `struct` but may add *inner* and *end padding* to fulfil *alignment requirements*.

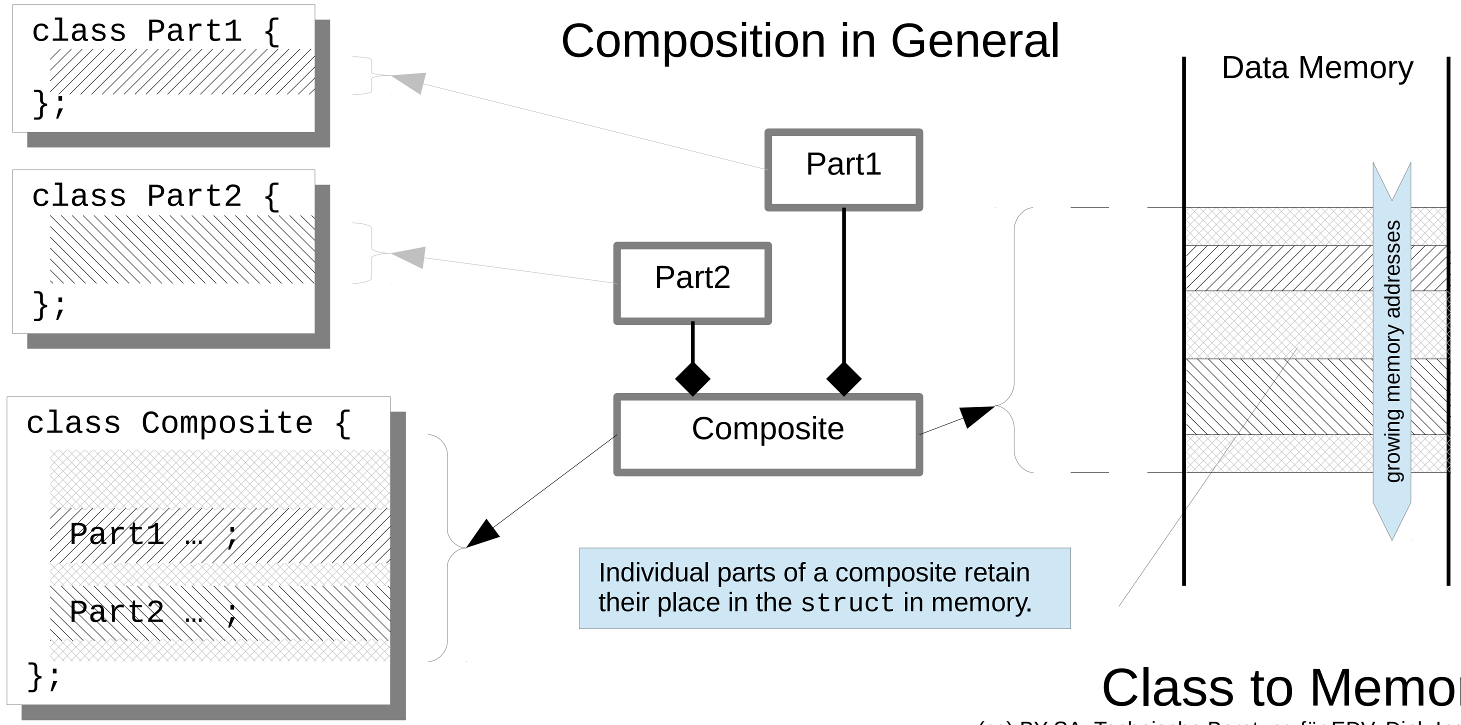
Ultimately combined and (re-) located to final addresses for branches in the code by the *Linker* or *Loader* (tool).

typcally stored in a single *Relocatable Object File* per *Compilation Unit* (e.g. *`modname`*`.cpp` compiles to *`modname`*`.o` or *`modname`*`.obj`) which may optionally be made part of some

- *Static Library* (e.g. *`libname`*`.a` or *`libname`*`.lib`) using a *Librarian* (tool) or
- *Dynamic Library* (*`libname`*`.dll`) aka. *Shared Object* (*`libname`*`.so`) using the *Linker* (tool)

Unique addresses ultimately assigned by the *Linker* or *Loader* (tool).

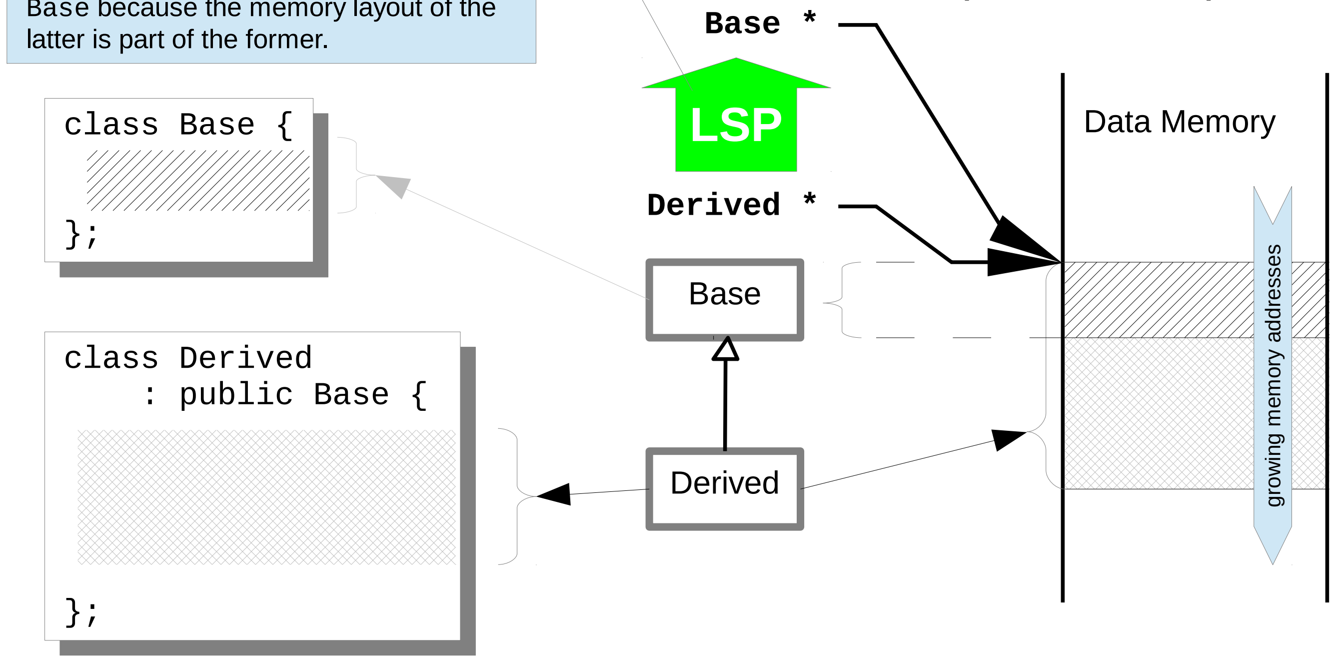
Composition in General



Individual parts of a composite retain their place in the `struct` in memory.

Public Base Class (Inheritance)

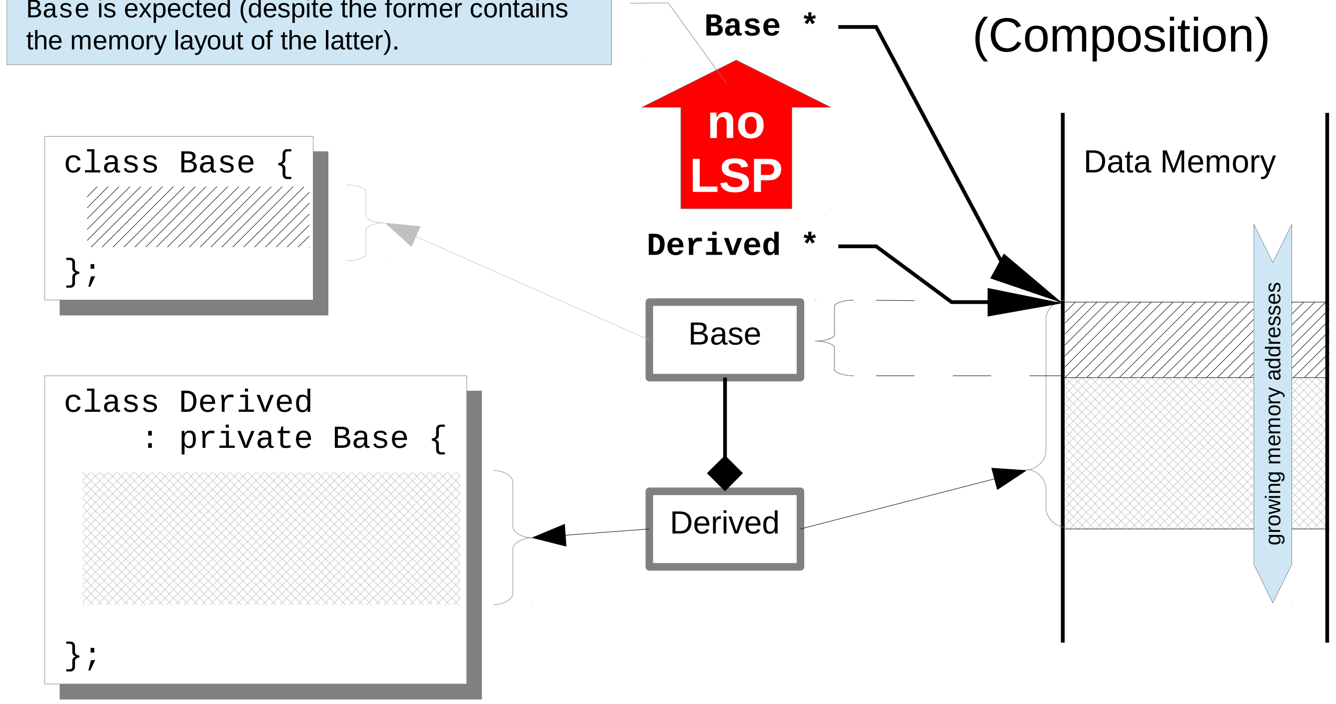
The LSP is part of the type conversion rules built-in to the compiler (and a "no-op" at run-time for single ineritance): Any `Derived` can trivially be used as a `Base` because the memory layout of the latter is part of the former.



… versus …

Privat Base Class (Composition)

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).



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

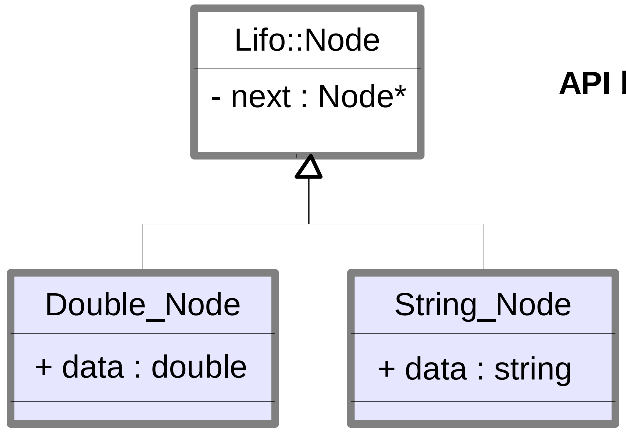
- Any object of a derived class should be a valid substitute for an object of its – direct or indirect – base classes.

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.
For private base classes there is no LSP in C++, hence they should be viewn as *Composition*, not *Inheritance*!

Class to Memory Mapping

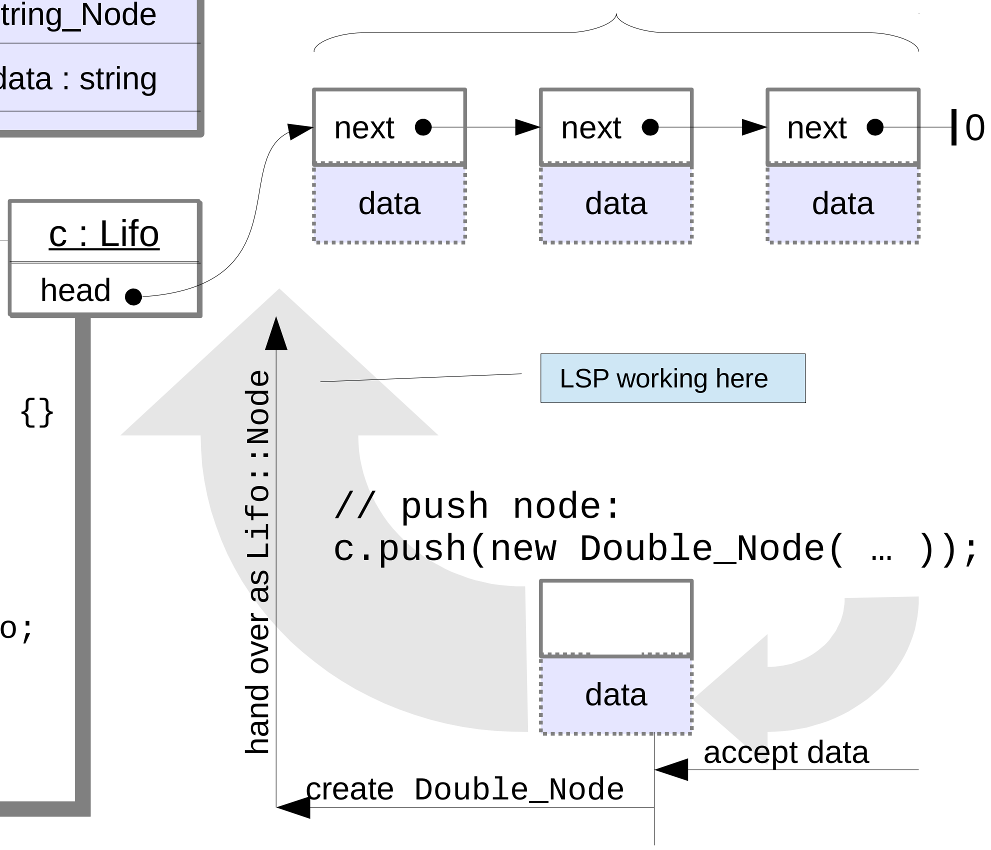
(Node-) Inheritance Technique

API based on pointers and thus implicitely polymorphic



Stored Elements:
- formally `Lifo::Node`-s
- actually derived classes

```
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    push(Node *);
    Node *pop();
};

class Lifo::Node {
    friend class Lifo;
    Node* next;
};
…
Lifo c;
```




```
// push node:
c.push(new Double_Node( … ));
```

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

take data

extend to Double_Node*

get `Lifo::Node*`

dynamic down-cast may return `nullptr`

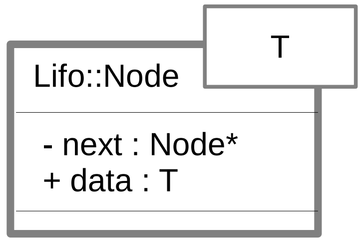
unexpected node types may cause memory leak with this coding style!

```
// 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 …
```

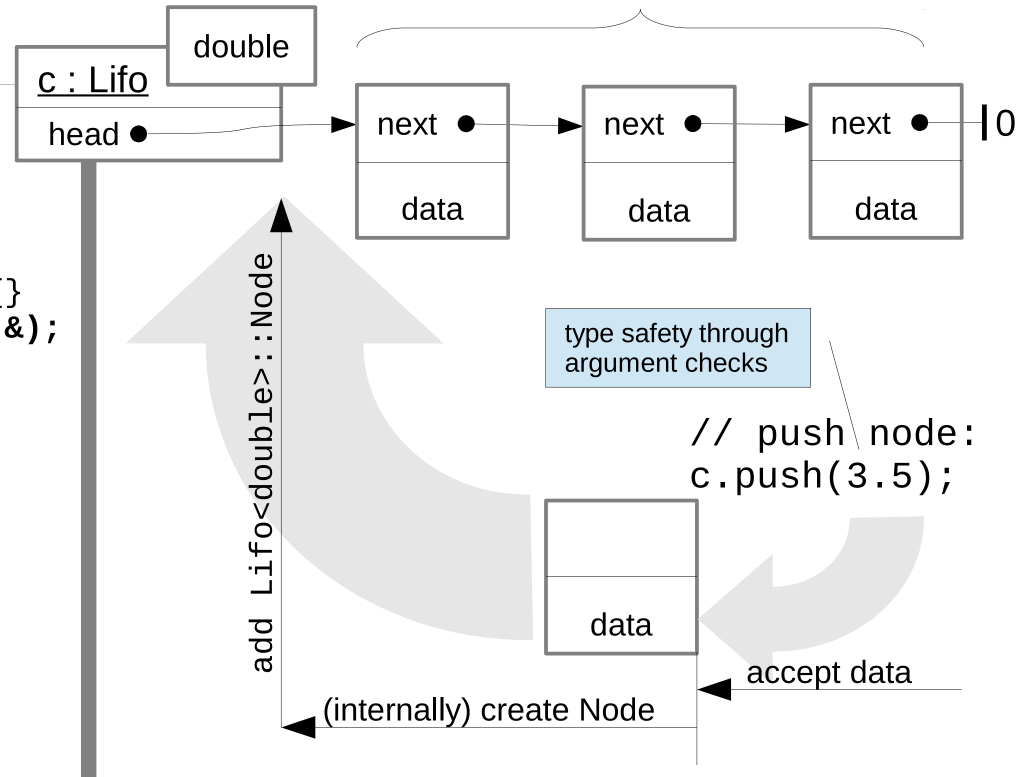
Template Technique

API based on values and thus non-polymorphic by default



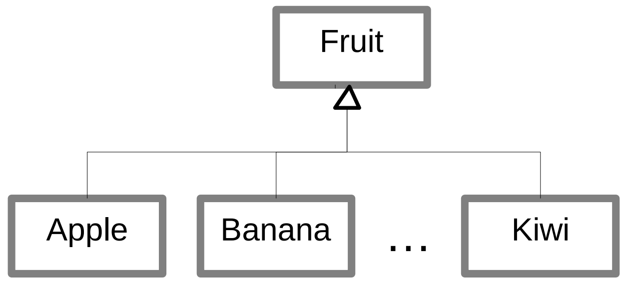
Stored Elements:
- `Lifo<double>::Node-s`

```
template<typename T>
class Lifo {
    class Node;
    Node *head;
public:
    Lifo() : head(0) {}
    void push(const T &);
    bool pop(T &);
};

template<typename T>
class Lifo::Node {
    friend class Lifo;
    Node* next;
public:
    T data;
};
…
Lifo<double> c;
// or:
// Lifo<std::string> …
```




```
// push node:
c.push(3.5);
```

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



Polymorphic Elements

for polymoprhic elements containers must use pointers explicitely

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

now points to an `Apple`

Non-Polymorphic Elements

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

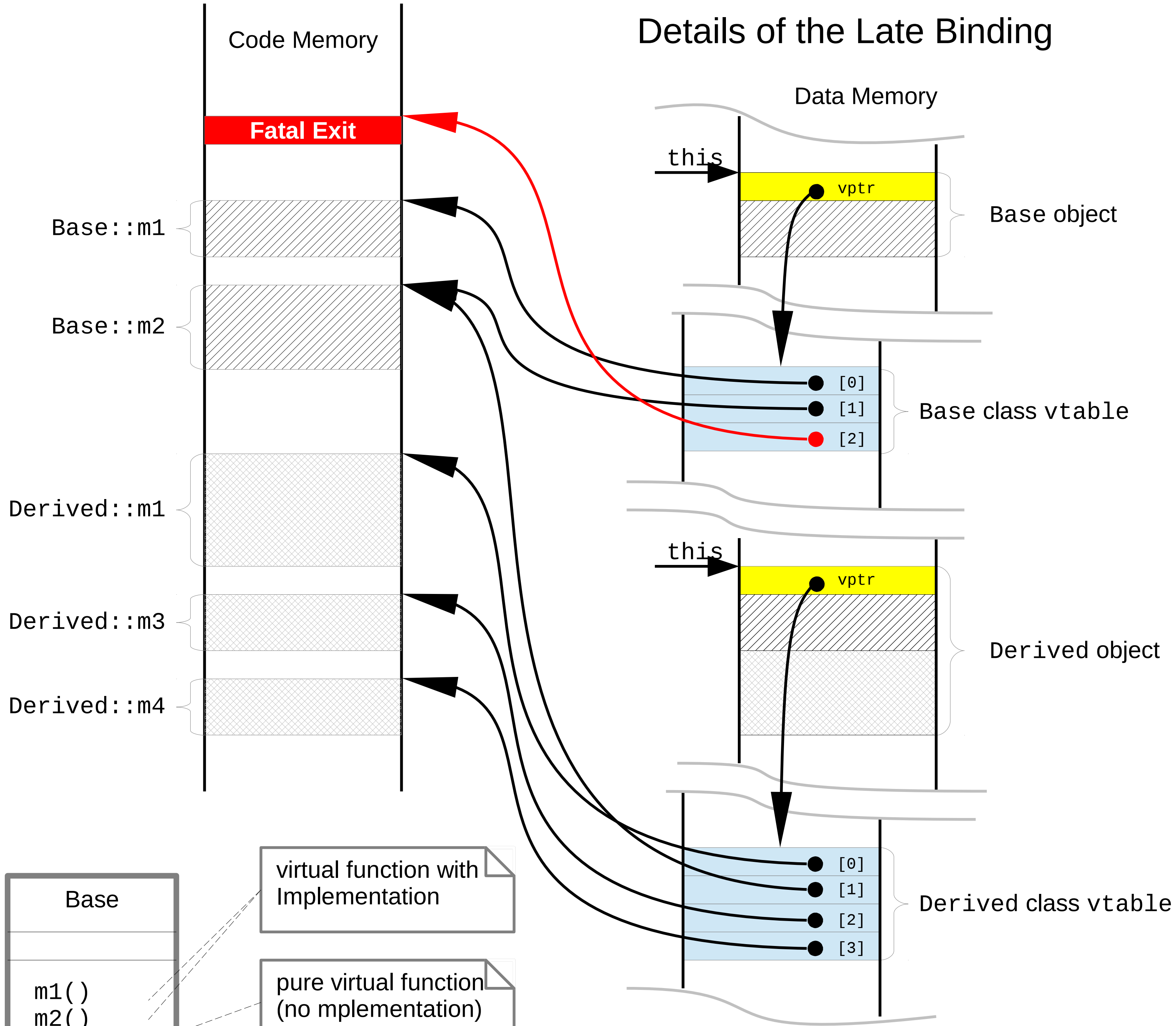
generallly compiles but will slice `Apple`-s and `Banana`-s to their `Fruit` base

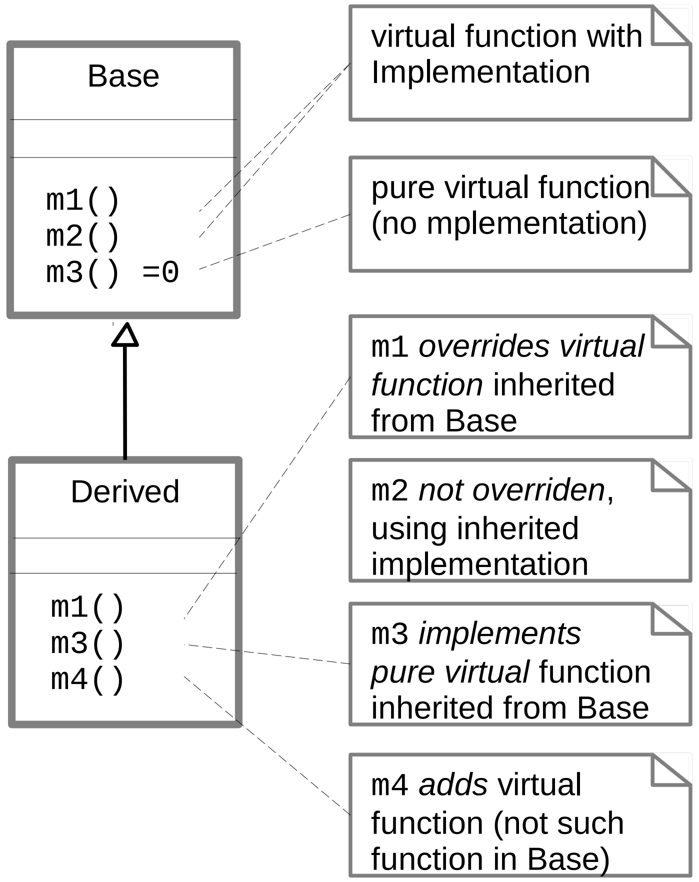
OK but, just a `Fruit` … (Sorry Sir, doesn't taste like a Banana anymore)

generally compiles but propably does not what is expected because only the `Fruit`-part of the apple gets modified

Container Implementation Techniques

Details of the Late Binding





Virtual function call in detail:

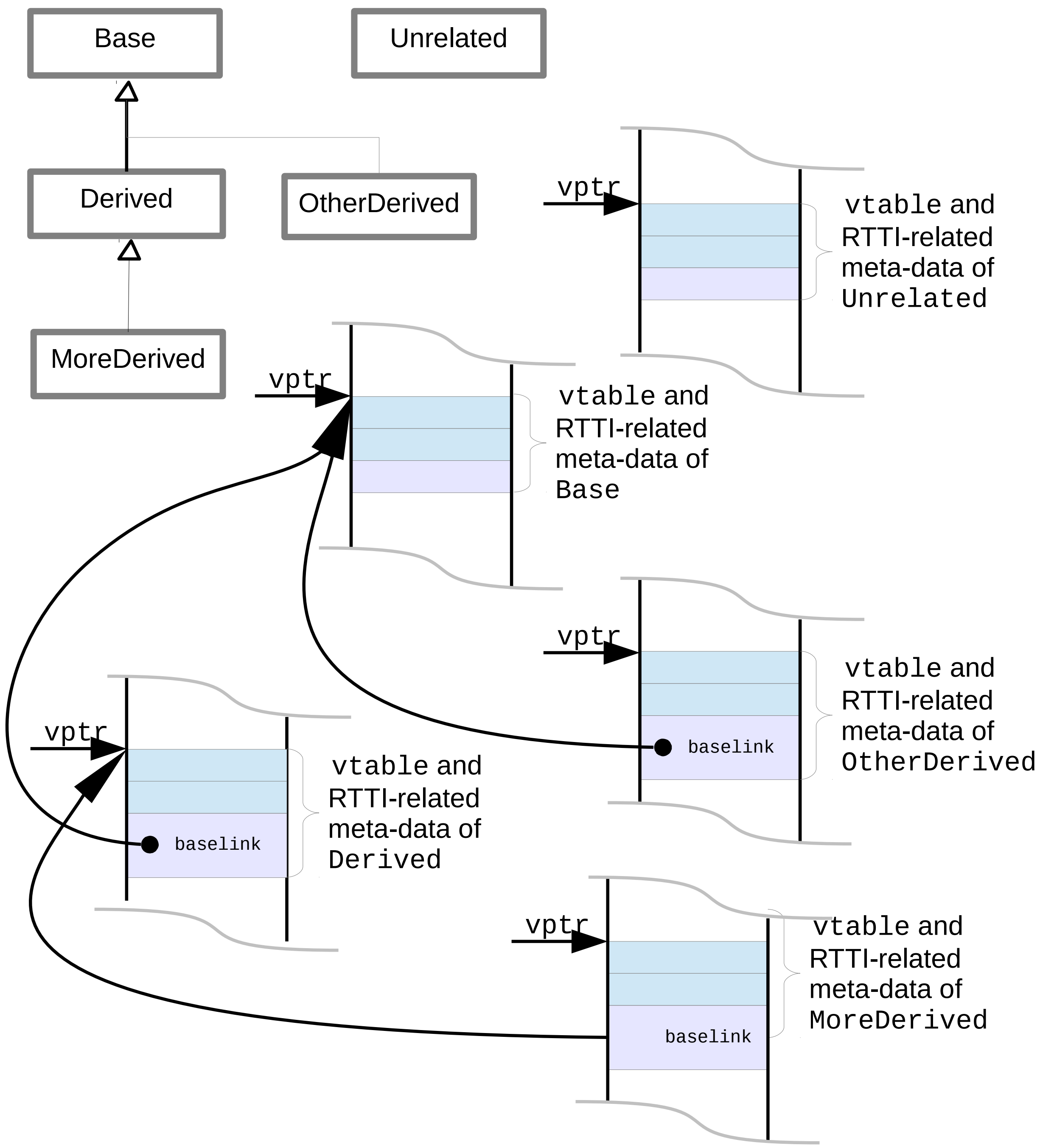
- starting from the `this` pointer
- locate the `vtable`[1] via the `vptr`[2]
- add member function specifiy offset
- read address from code segment
- branch to this location as sub-routine

—

[1]: `vtable`-s exist **one per class** (with at least one virtual member function)

[2]: `vptr`-s exist **one per object** (of a class with at least one virtual member function)

Storing RTTI-Related Meta-Data



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

- This avoids overhead which would otherwise occur per object.
- Meta-data is stored in the vincinity of (or linked with) 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

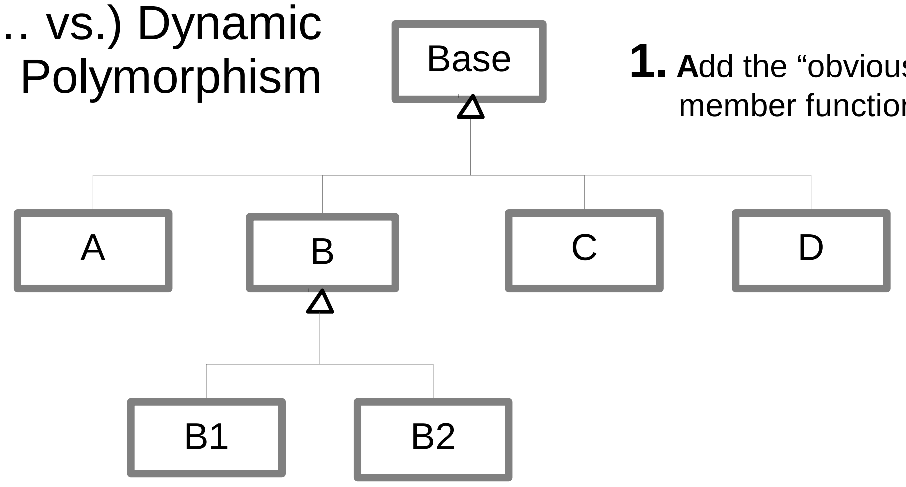
```
void foo(Base &r) {
    …
    if (auto p = dynamic_cast<A*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<B1*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<B2*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<C*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<D*>(&r)) {
        …
    }
    …
}
```

```
    …
    // combining B1 and B2
    if (auto p = dynamic_cast<B*>(&r)) {
        …
    }
    …
```

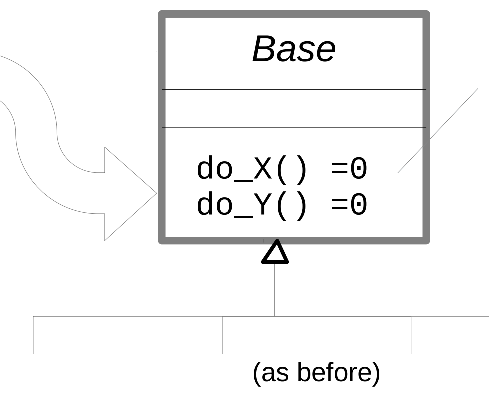
```
void foo(Base &r) {
    …
    if (typeid(r) == typeid(A)) {
        …
    }
    if (typeid(r) == typeid(B1)) {
        …
    }
    if (typeid(r) == typeid(B2)) {
        …
    }
    if (typeid(r) == typeid(C)) {
        …
    }
    if (typeid(r) == typeid(D)) {
        …
    }
    …
}
```

```
    …
    // combining B1 and B2
    if (typeid(r) == typeid(B1)
     || typeid(r) == typeid(B2)) {
        …
    }
    …
```

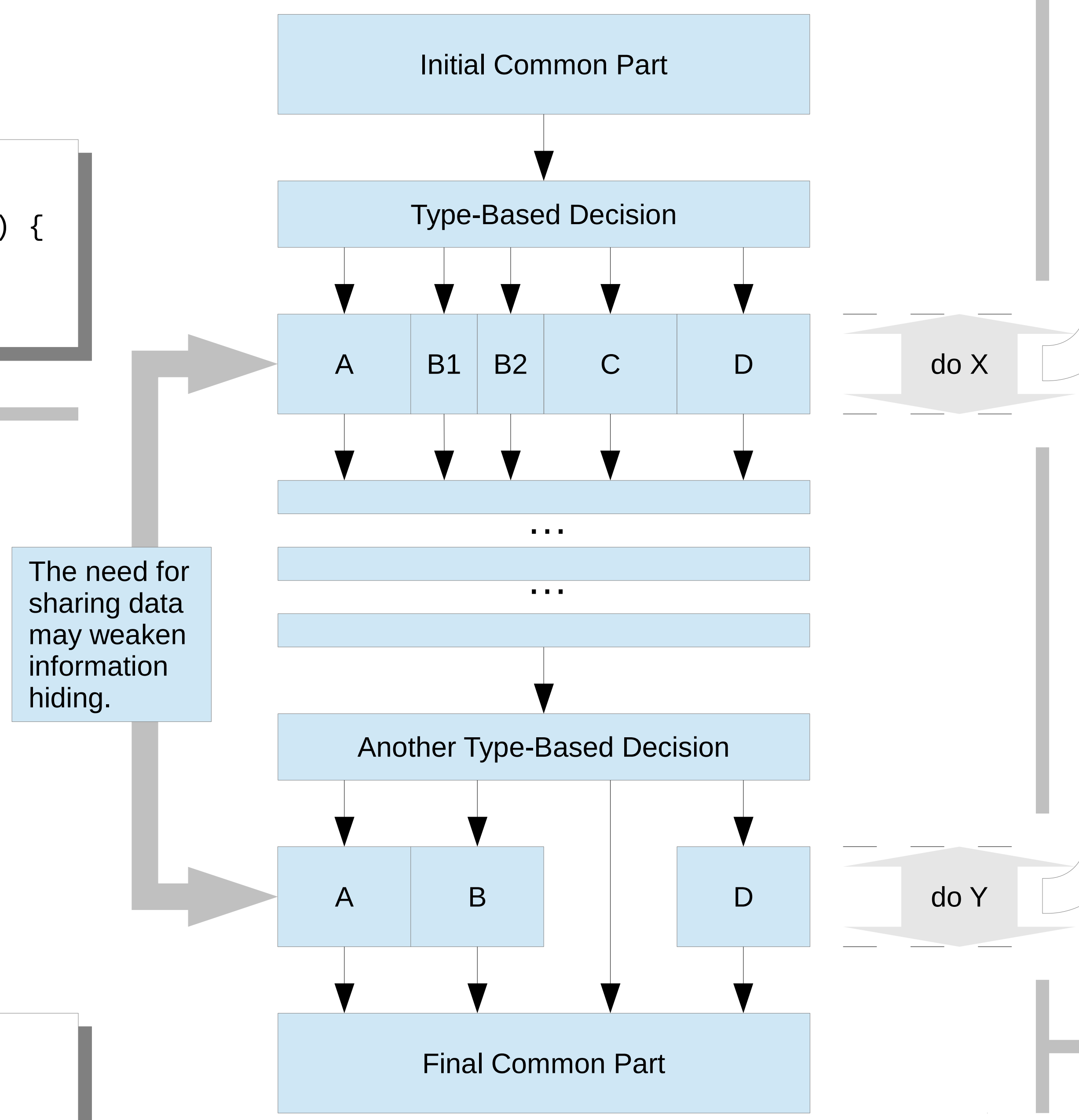
(… vs.) Dynamic Polymorphism



1. Add the "obviously missing" member functions to `Base`:



Alternatively common defaults may be provided here.



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() {
    …
}
```

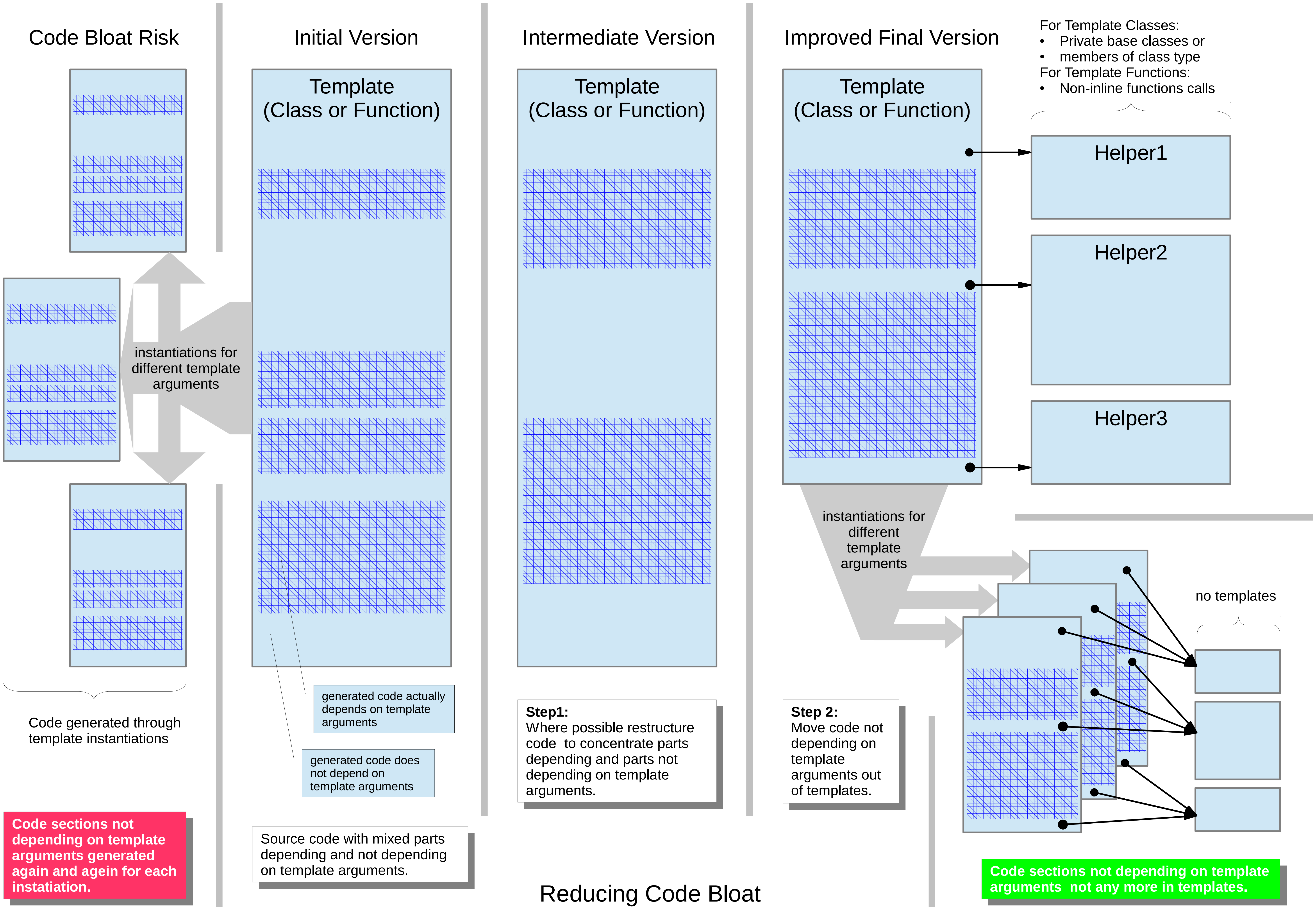
```
void A::do_Y() {
    …
}
void B::do_Y() {
    …
}
void C:: do_Y() {
    /*empty*/
}
void D:: do_Y() {
    …
}
```

Data can be shared easily in privacy.

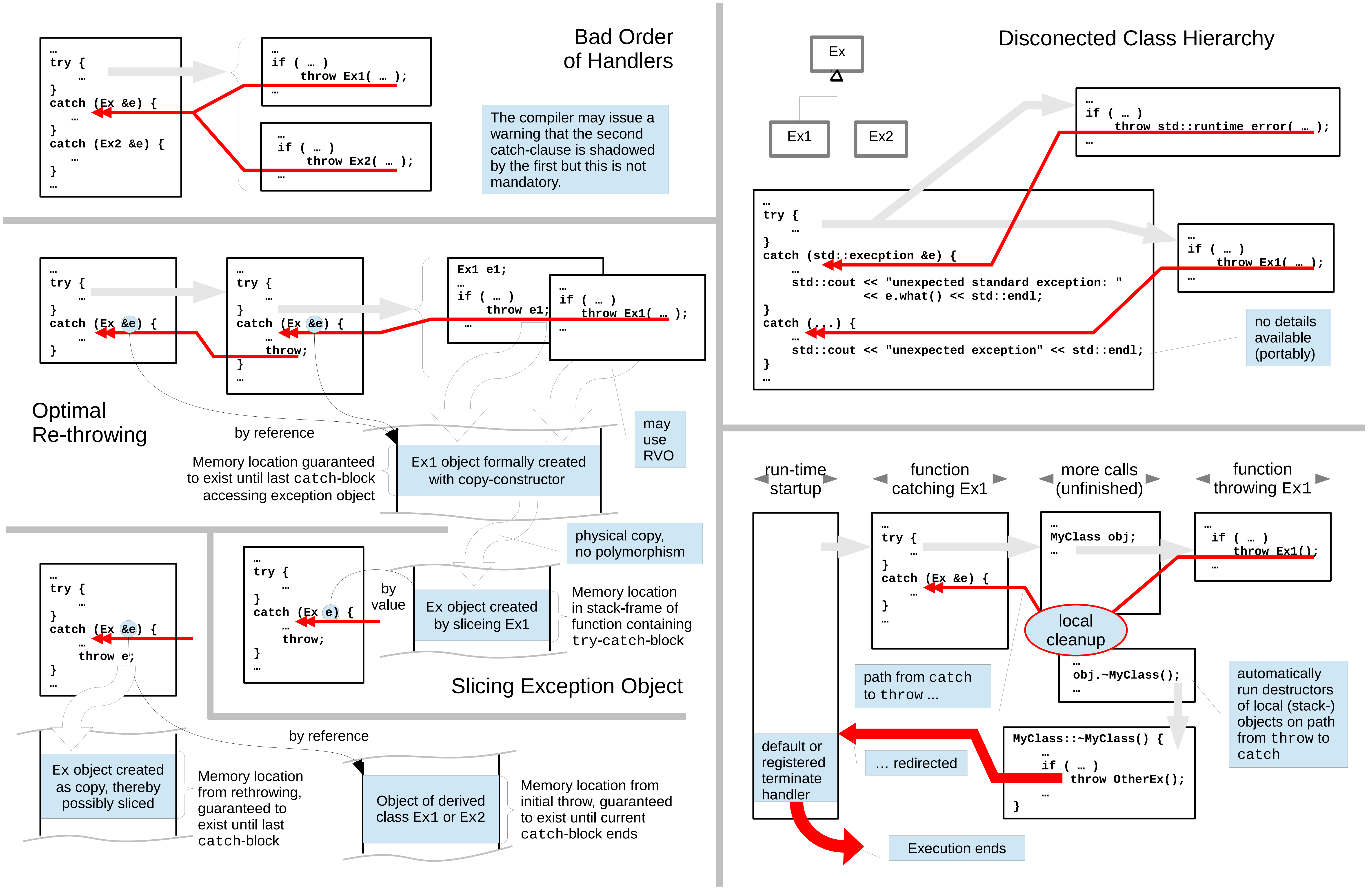
3. Replace multiway branches with member function calls:

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

Type-Based Multiway Branching



Reducing Code Bloat



Sub-optimal Re-throwing

Throwing from Destructors

Exception Details

Classic Resource Management APIs

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

Turn into RAII

Resource Handle

```
class FileRes {
    File *fp;
    …
};
```

Acquisition

```
FileRes::FileRes(
    const char n[],
    const char m[]
) : fp(fopen(n, m)) {}
```

Release

```
FileRes::~FileRes() {
    fclose(fp);
}
```

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;
}
```

Acquire Resource for Code Segment

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

```
void foo() {
    …
    if ( … )
        throw …;
    …
}
```

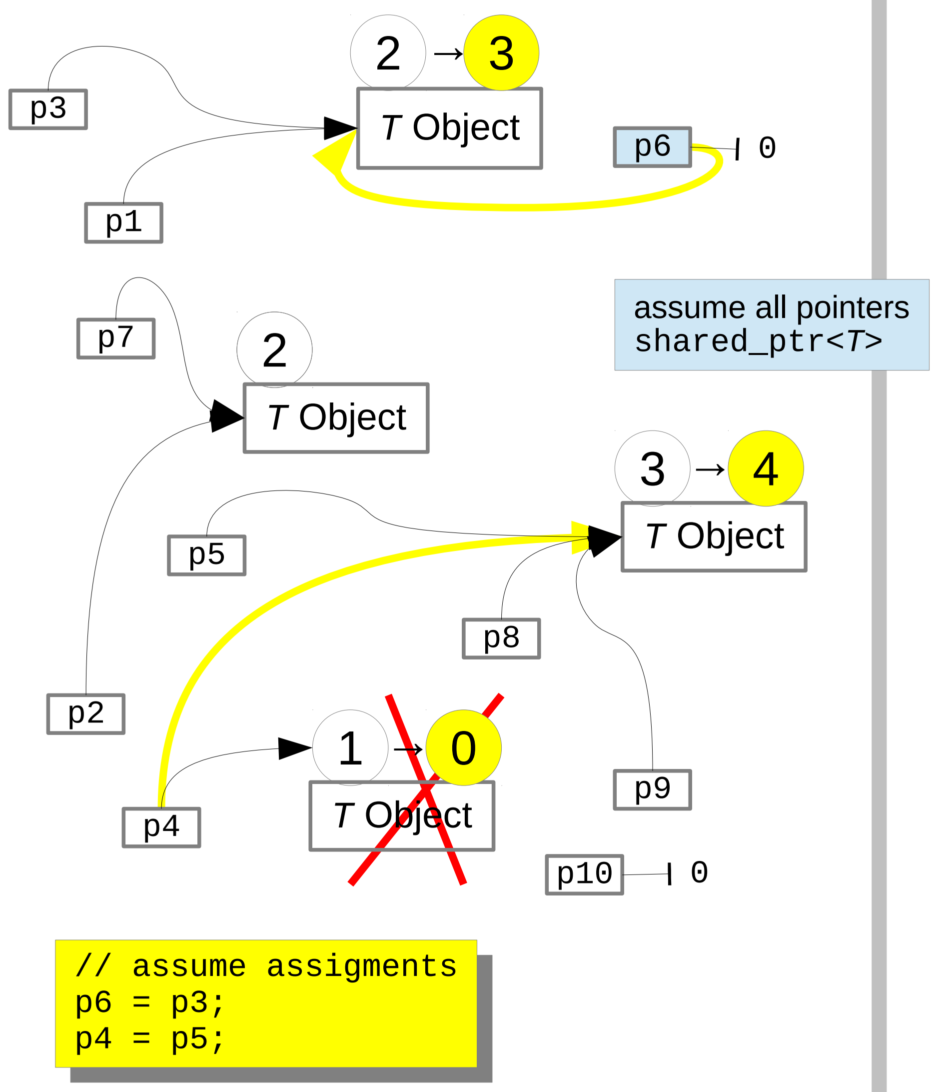
Acquire Resource for Object Lifetime

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

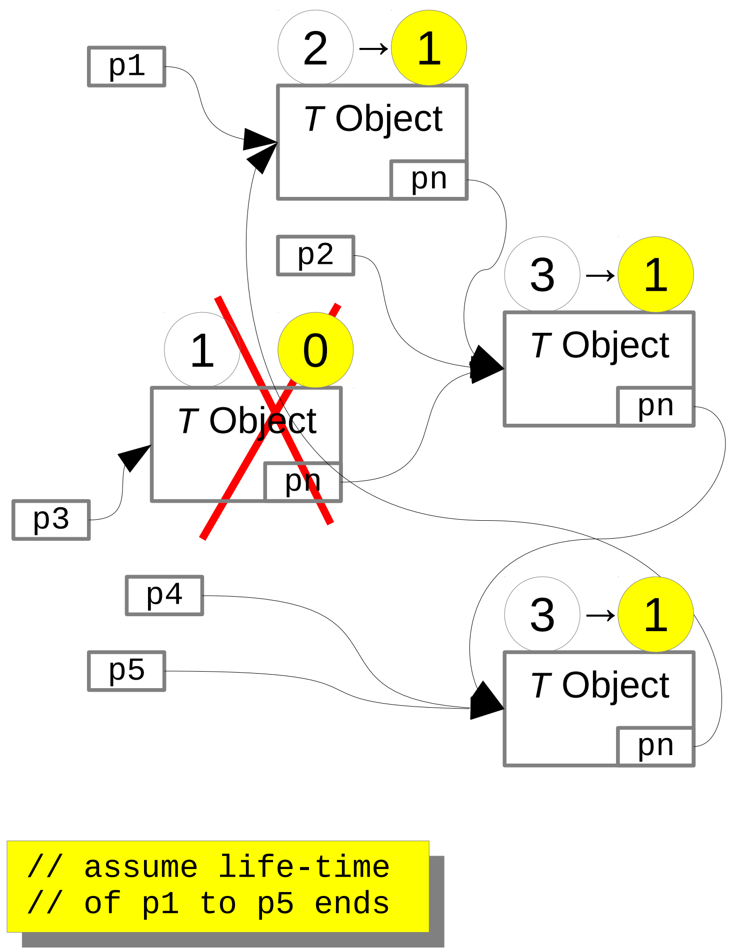
```
FileRes f( … );
…
char s[80];
fgets(s, sizeof s, f);
…
if (!ferror(f))
    …
```

Wrapped Resource

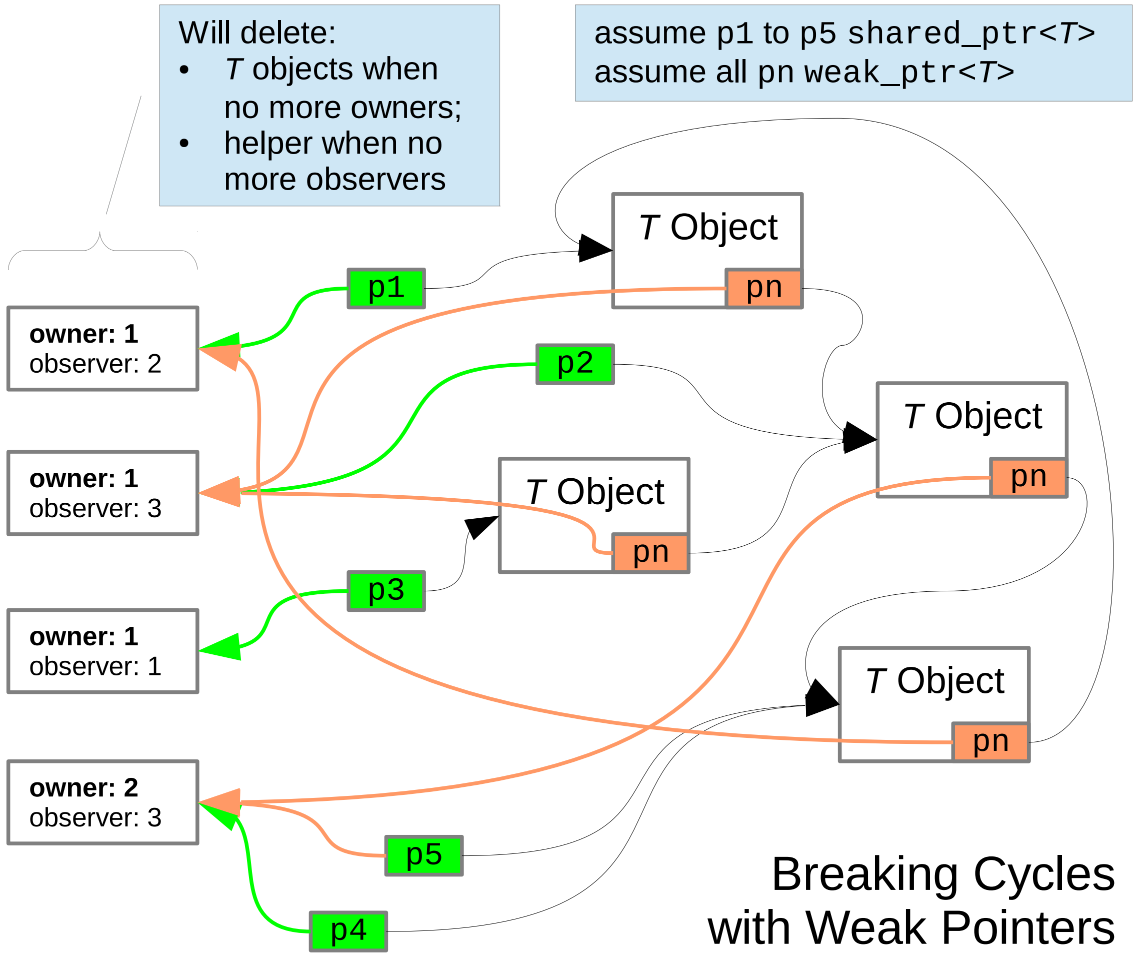
Classic Resource Management vs. RAII



Reference Counting Principle

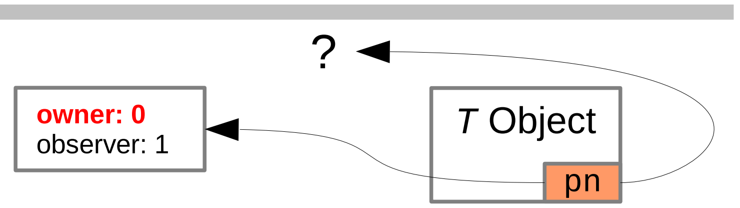


Problem of Cyclic References



Breaking Cycles with Weak Pointers

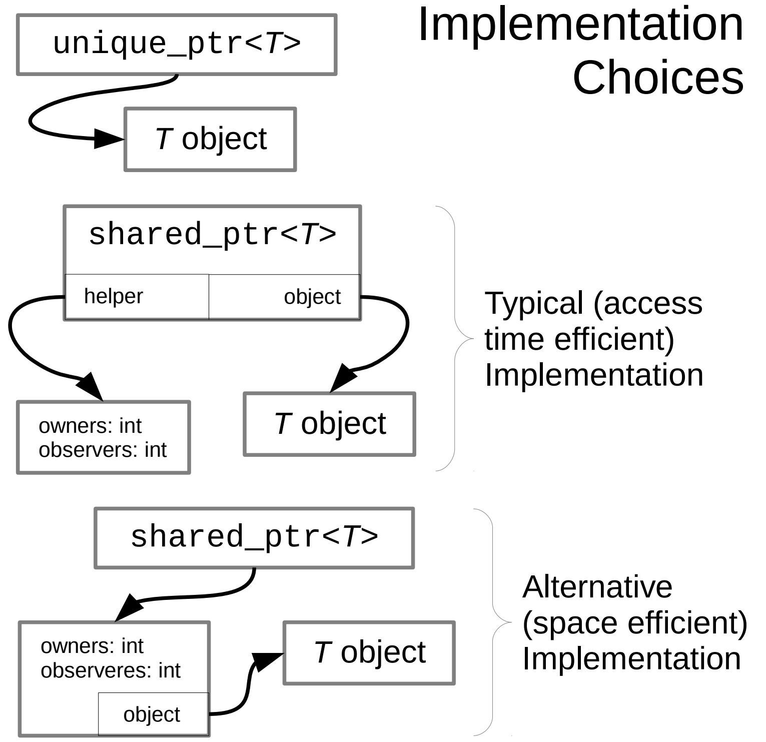
Dangling Weak Pointer



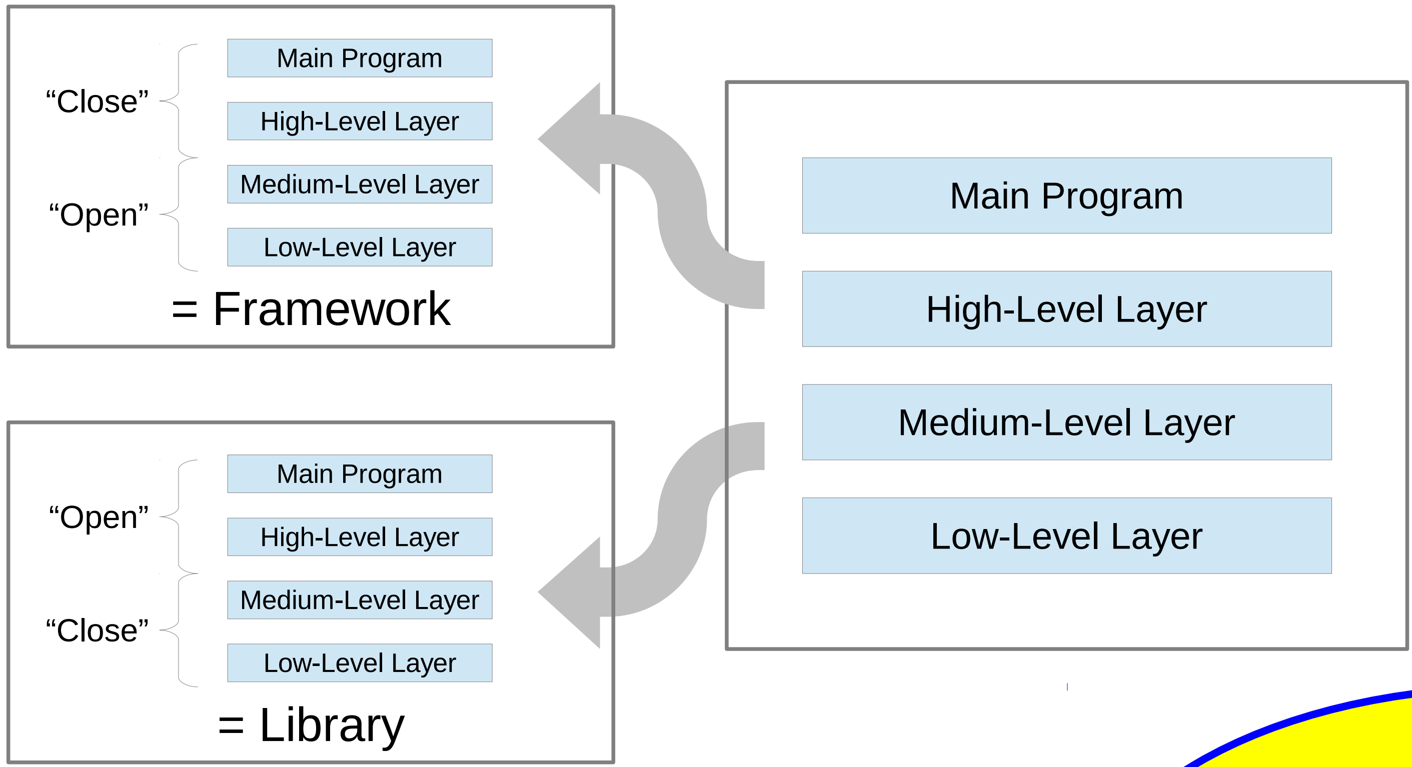
Comparing ...	`std::unique_ptr<T>`	`std::shared_ptr<T>`	Remarks
Characteristic	refers to a single object of type *T*, **uniquely owned**	refers to a single object of type *T*, **possibly shared with other referrers**	may also refer to “no object” (like a `nullptr`)
Data Size	same as plain pointer	same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

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

Implementation Choices



Smart Pointer Comparison



Design for Reusability

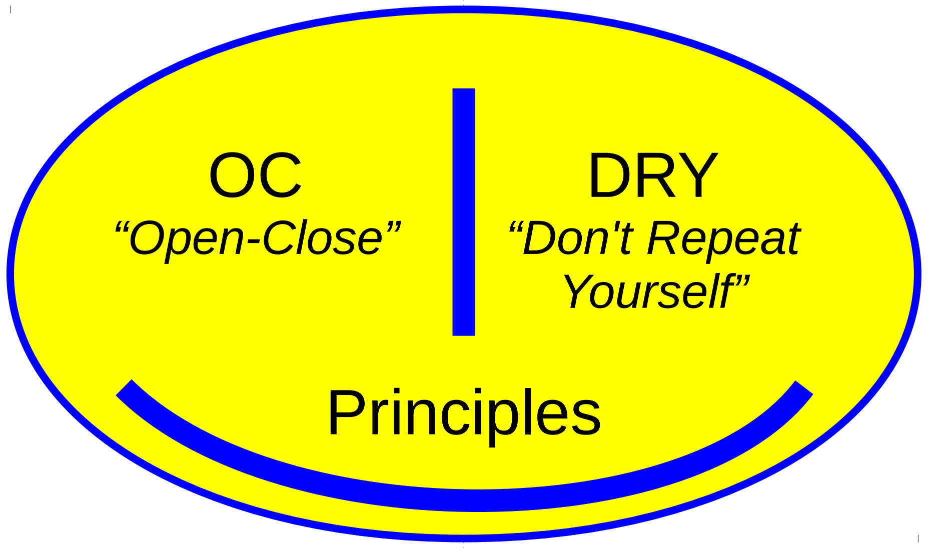
- *Libraries* or *Frameworks* for common components
- Classes for common services or abstractions
- C++-Templates for genericity in types

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



Parametrize for Flexibility with

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

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)

Apply Best Practices, e.g.

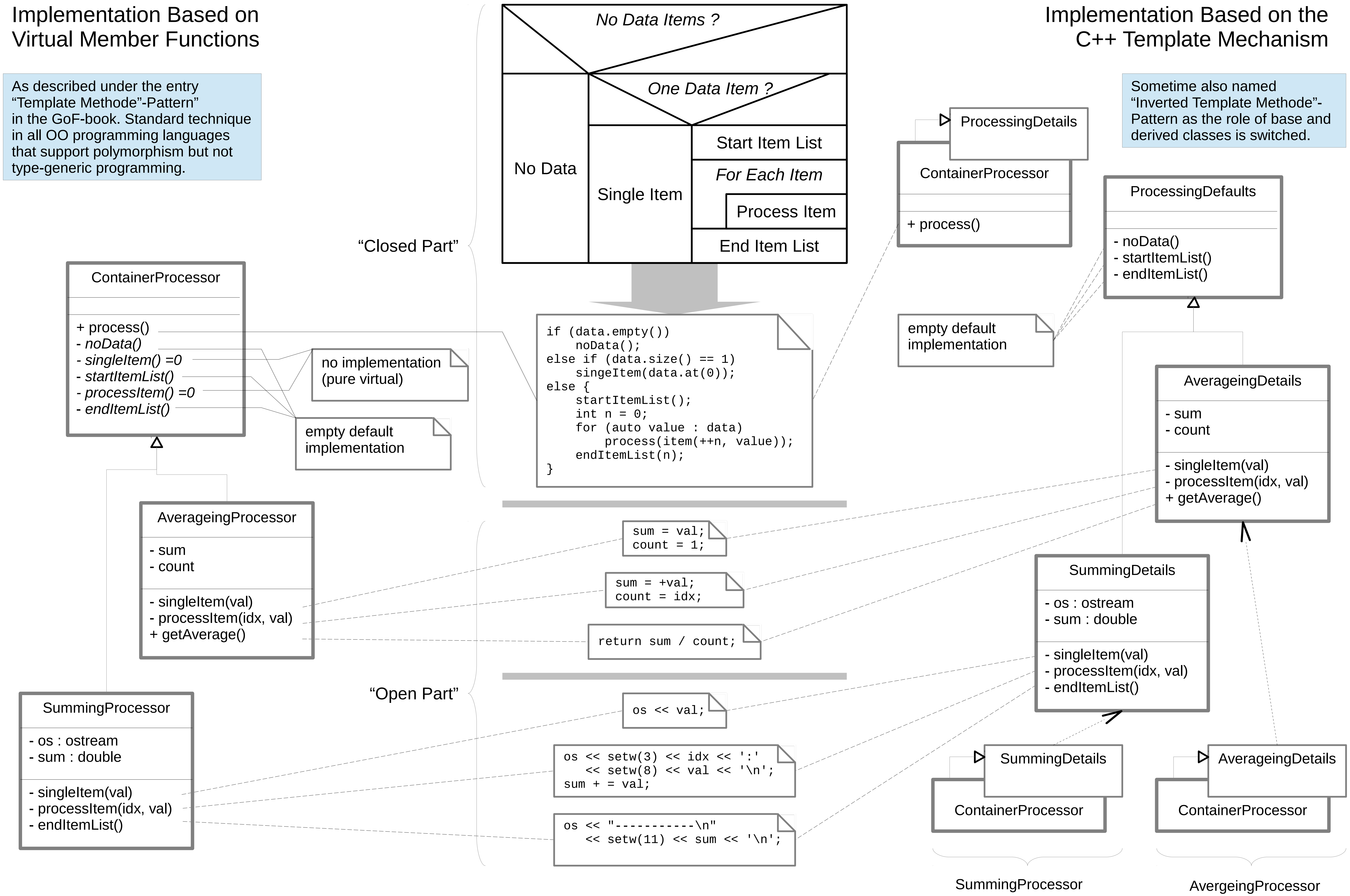
- Standard Design Patterns (from GoF) like
  - Composite
  - Template Methode
  - …
- Well-known C++ Idioms like
  - PIMPL (Pointer to Implementation)
  - RAII (Ressource Acquisition is Initialisation)
  - CRTP (Curiosly Recurring Template Pattern)
  - …
- Handy Little Techniques where useful
  - "Named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

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



Example – “Open Close”-Principle