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

Efficiency

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

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

C++11

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.

Classes

CharType
std::basic_string

Lookup in reference documentation here …

char
std::basic_string
`std::string`

wchar_t
std::basic_string
`std::wstring`

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

Numeric Conversions

convert to any builtin integral type

convert to any builtint floating point type

```
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) …
…
```

overloaded for any builtin integral and floating point type

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

Input and Output

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

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;
```

String Operations

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

Basic operations:
- construction, assignment, … (etc. as can be expected);
- single character access with
  - `operator[]` (unchecked)
  - or member function `at()` (throws for out-of-range);
- concatenation with `operator+` (`operator+=` for combined assignment).

Advanced operations:
- too many to list (→ RTFM).

```
string str1("HeLlO WoRld!");
to_upper(str1);
// str1=="HELLO WORLD!"
to_upper(str1);
// str1=="hello world!"
```

Regular Expressions

C++11

Providing yet another versatile and extremely powerful technique to ...

… **validate** a string for expected **content** (with `regex_match` and `regex_search`);

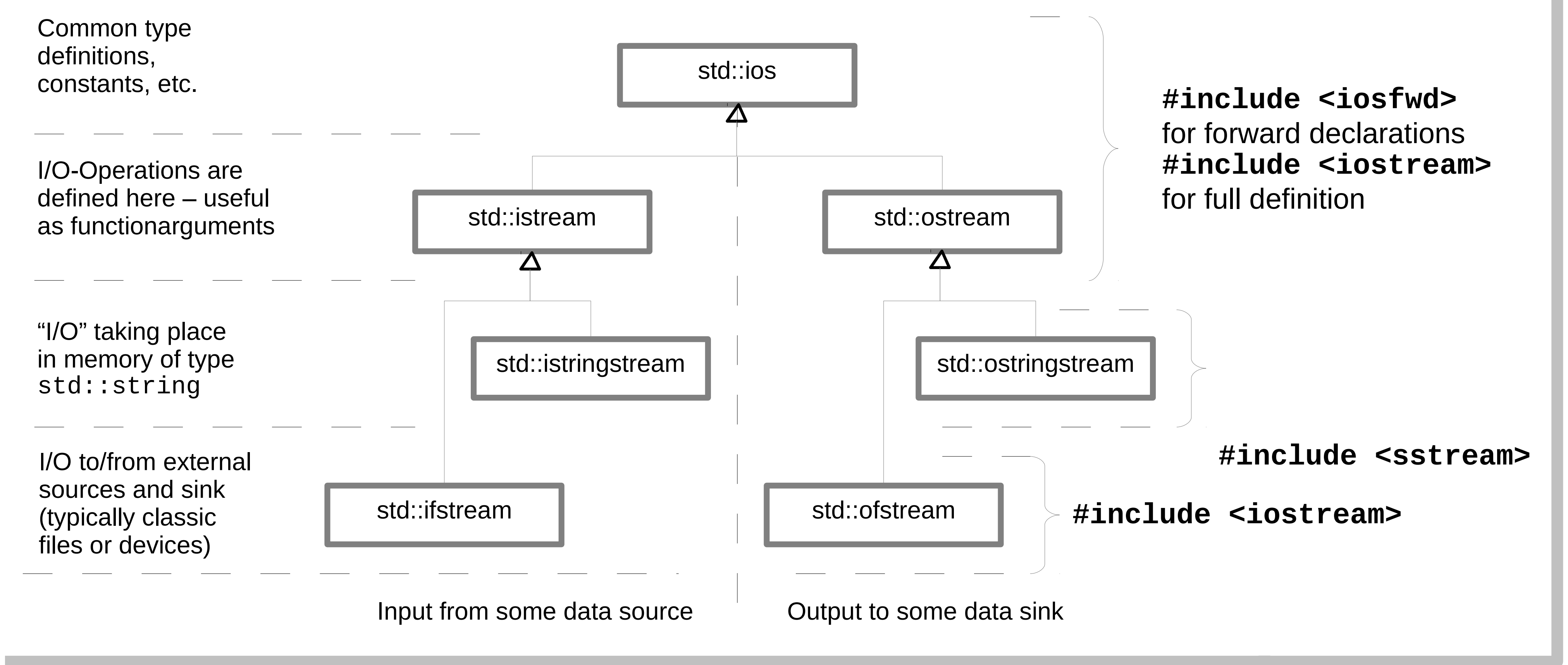
… **retrieve parts** from a string for further processing (with help of `match_results`);

… systematically **find and replace** textual content (with `regex_replace`).

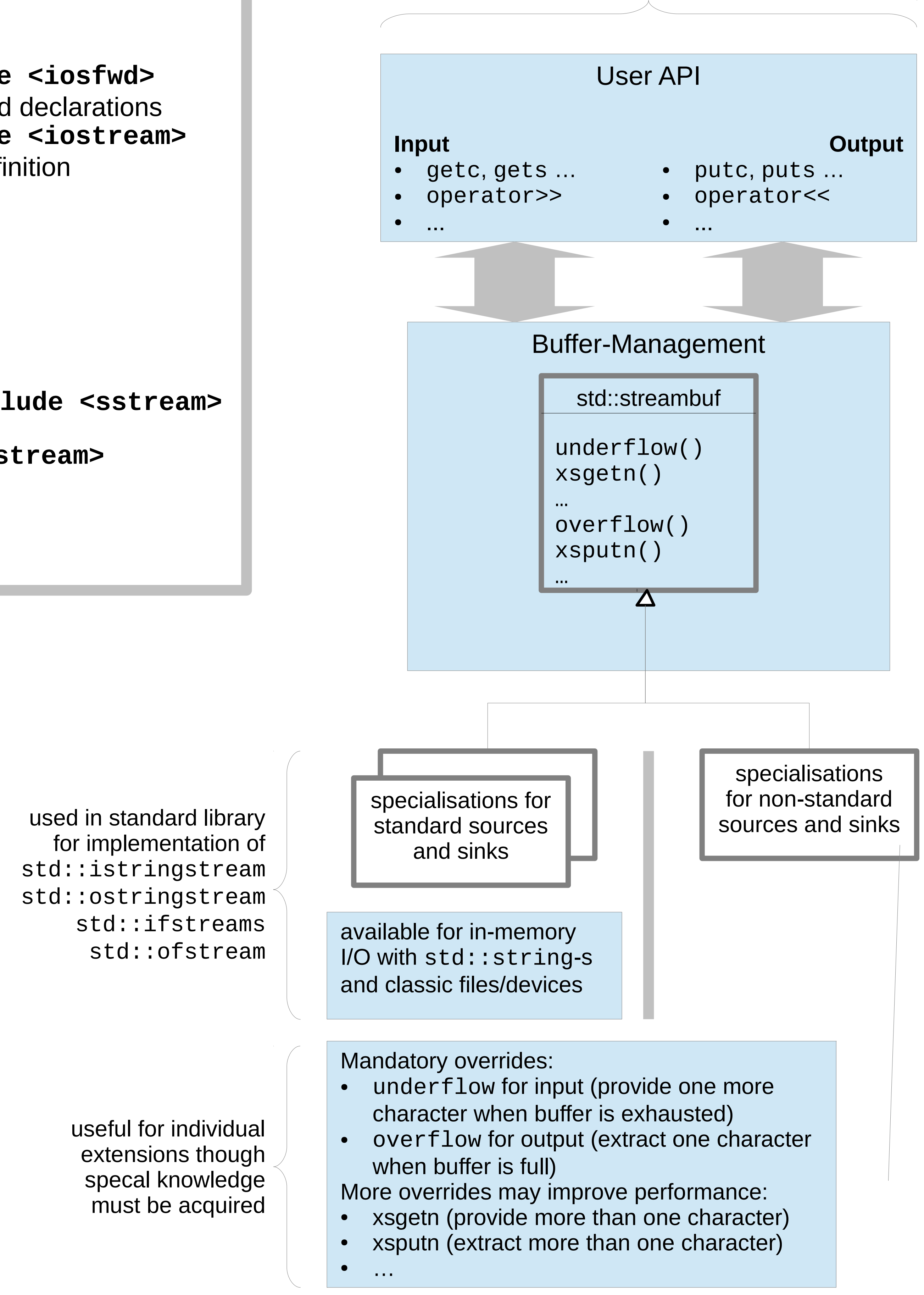
Boost Extensions

Boost.String_algo provides much more “seemingly missing” functionality for `std::string`-s e.g.
- remove white space (`trim`, `trim_left`, `trim_right`)
- parse into tokens (with `string_split_iterator`)
- join elements from a container (`join`, `join_if`)
- ...

I/O-Streams Front-End



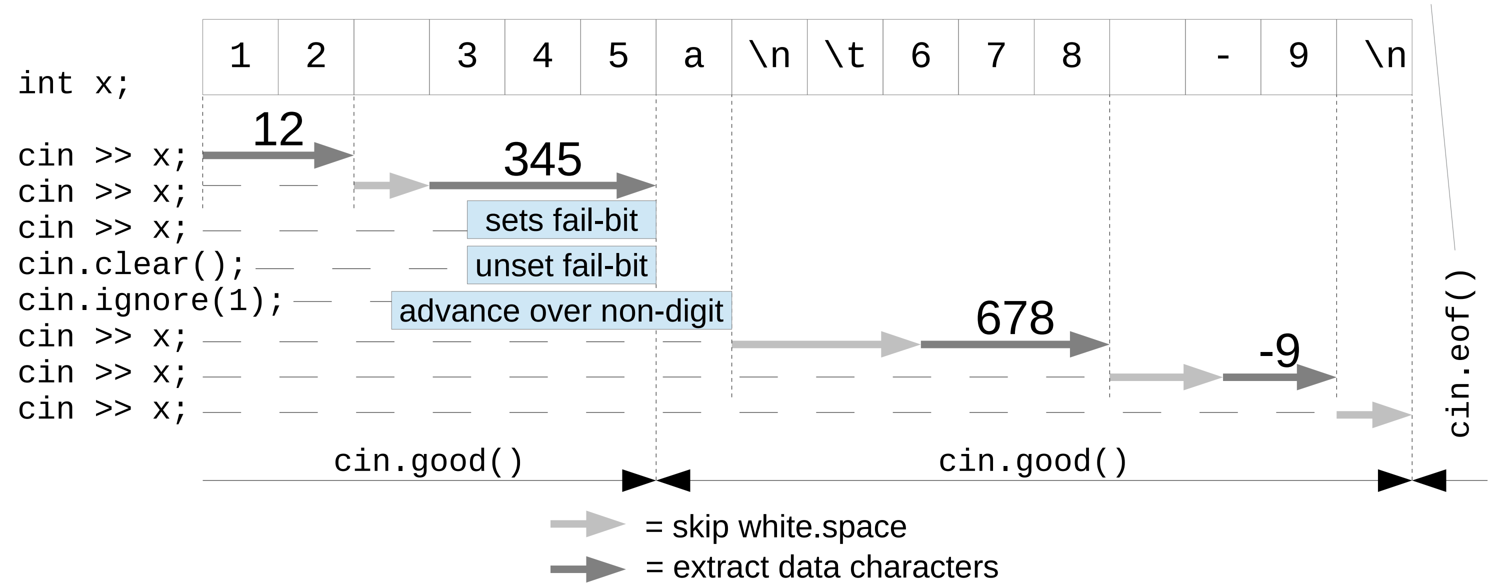
"day to day" use of C++



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)		

For keyboard input use: CTRL-D (Unix) or CTRL-Z (DOS)



I/O-Streams State-Bits

I/O-Stream Basics

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(size() > offset);
     const std::size_t idx = (iput >= (offset+1))
                    ? iput - (offset+1)
                    : iput + 11 - (offset+1);
     return data[wrap(idx)];
}
```

```
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;
```

Parametrizing Size

```
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 {
        assert(size() > offset);
        const std::size_t idx = (iput >= (offset+1))
                                ? iput - (offset+1)
                                : iput + (Size+1) - (offset+1);
        return data[wrap(idx)];
}
```

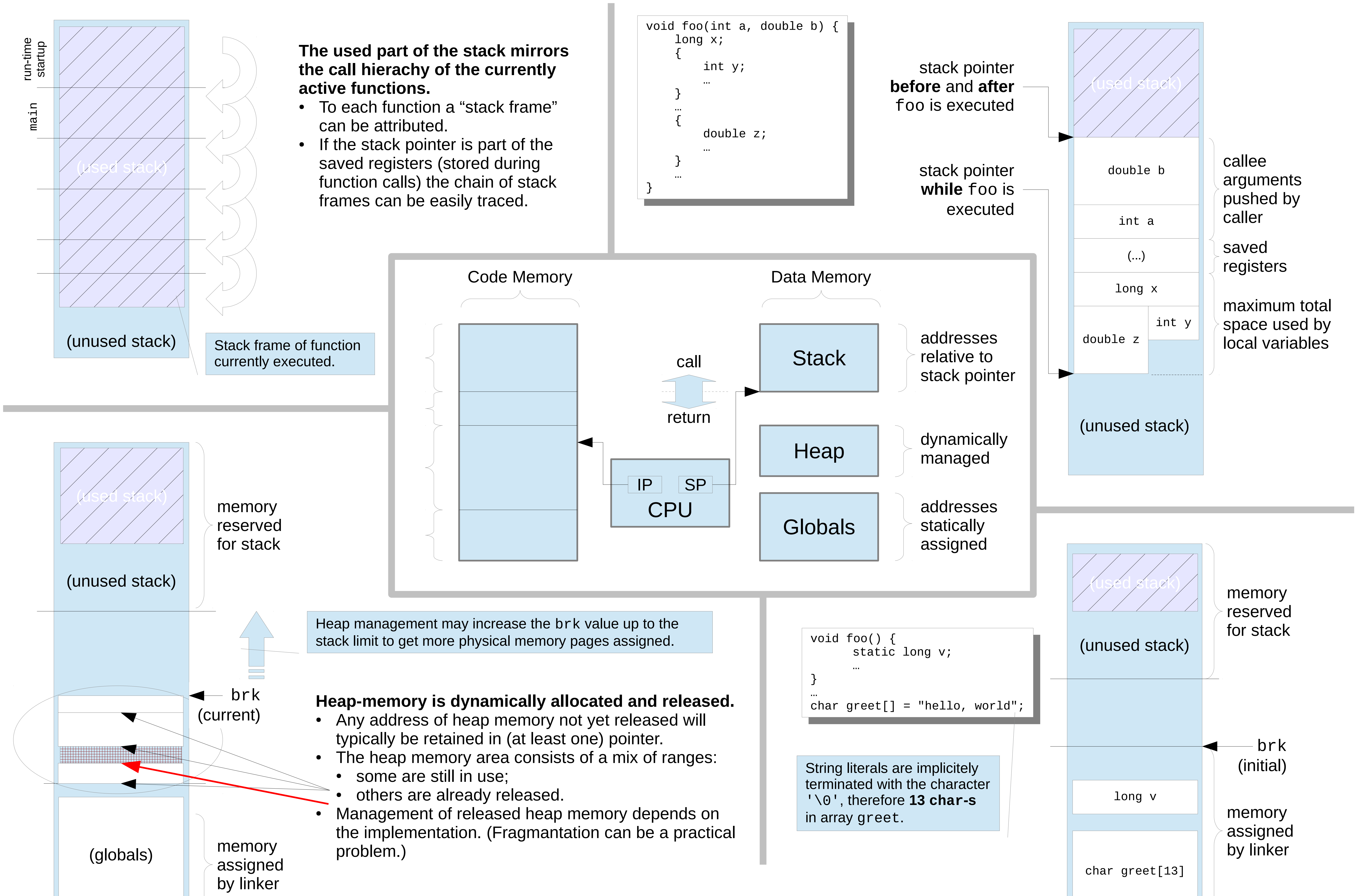
```
RingBuffer<10> b;
```

Instantiations:

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

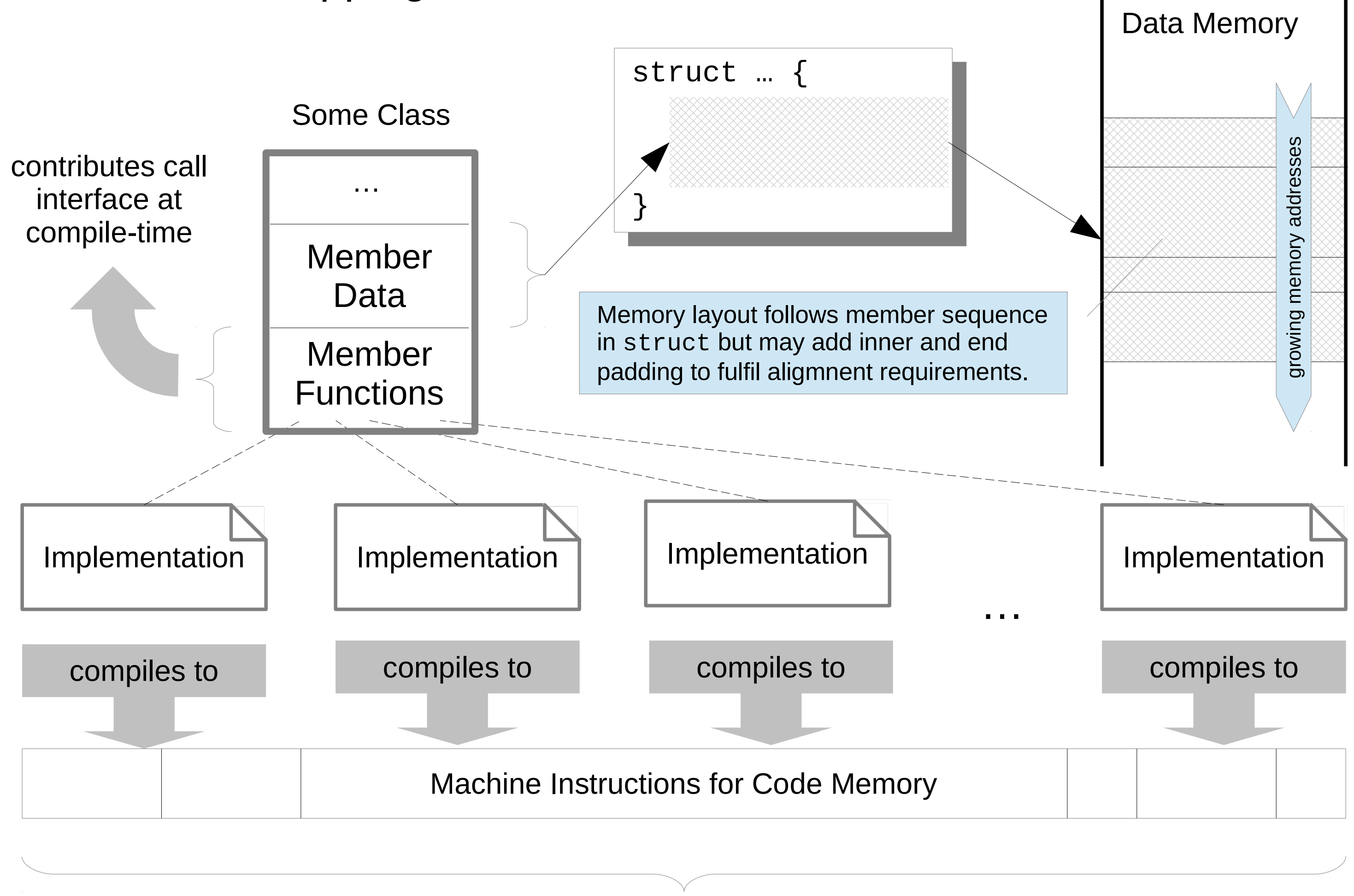
```
template<typename T, std::size_t N>
class RingBuffer {
     double data[N+1];
protected:
     std::size_t iput;
     std::size_t iget;
     static std::size_t wrap(std::size_t idx) {
          return idx % (N+1);
     }
public:
     RingBuffer()
          : iput(0), iget(0)
     {}
     bool empty()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

Parametrizing Types and Sizes



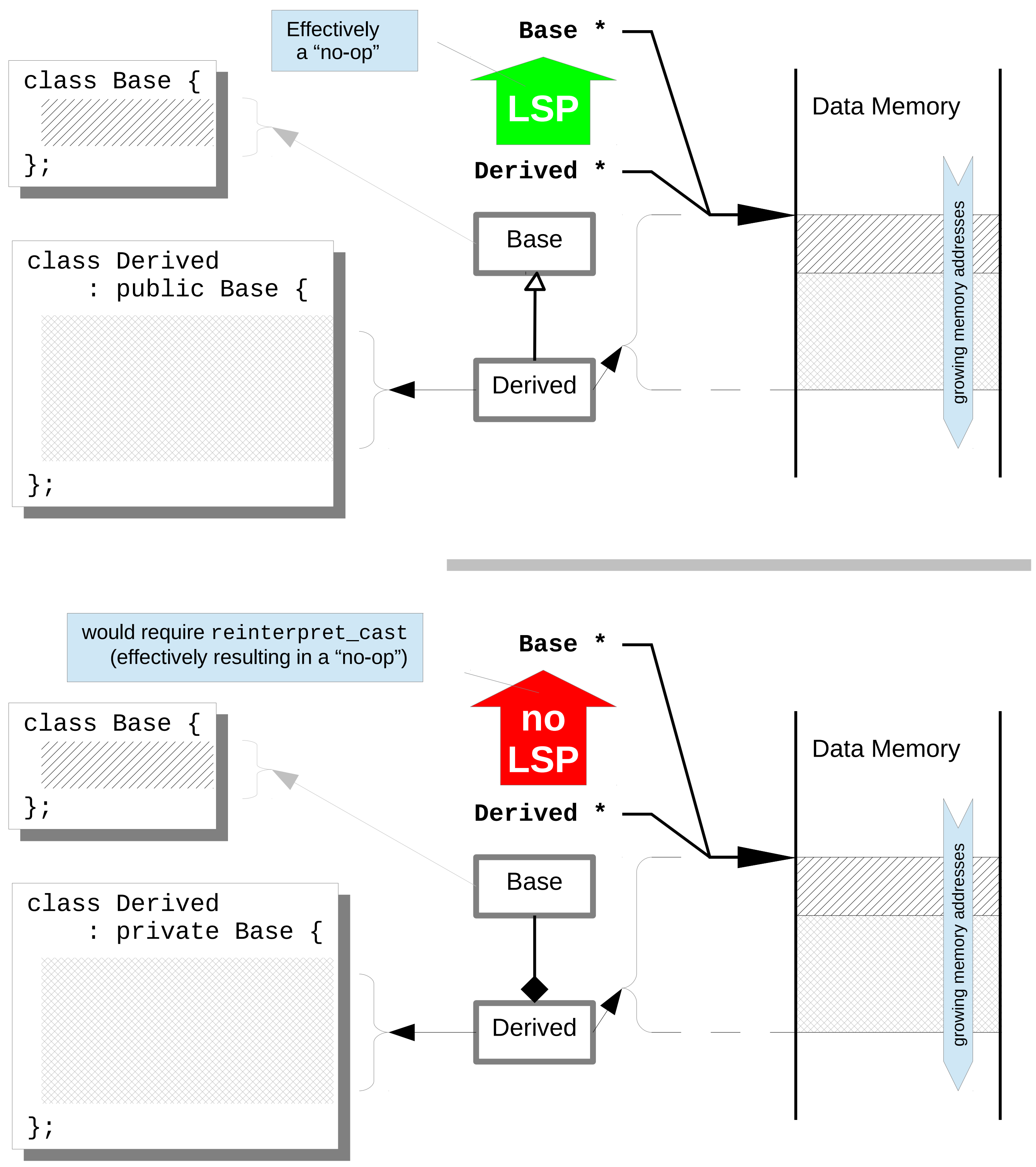
Generalised Execution Model

Mapping Classes to Code And Data

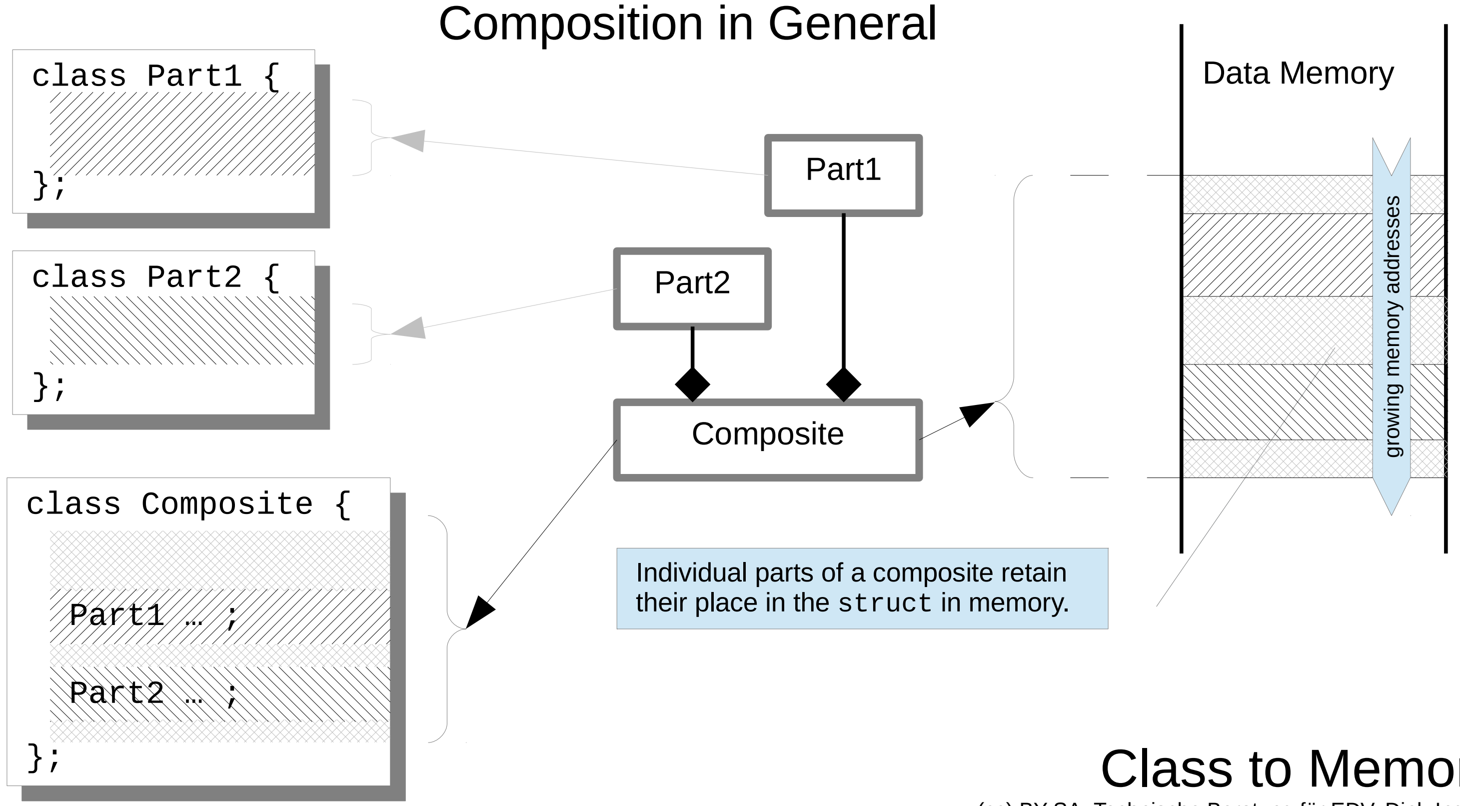


In case of individual separate compilation and results stored in static libraries the linker may include or omit object modules as necessary to for the final executable program.

Public versus Privat Base Classes



Composition in General



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

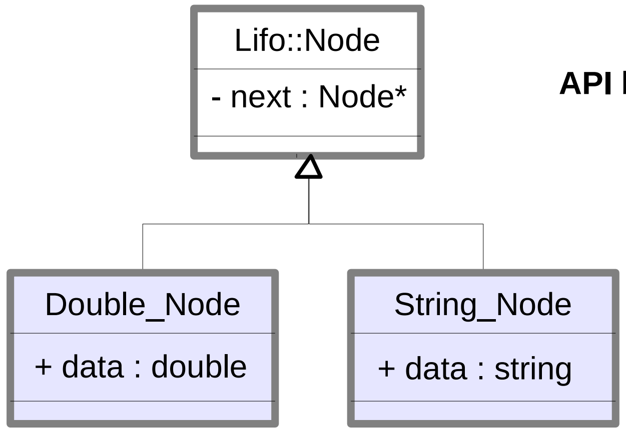
- Any object of a derived class should be a valid substitute for an object of its (direct or indirect) base classes.
- While only single inheritance is used the LSP is effectively a "no-op" in C++ since base class objects start at the same memory address as their derived classes.

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

Class to Memory Mapping

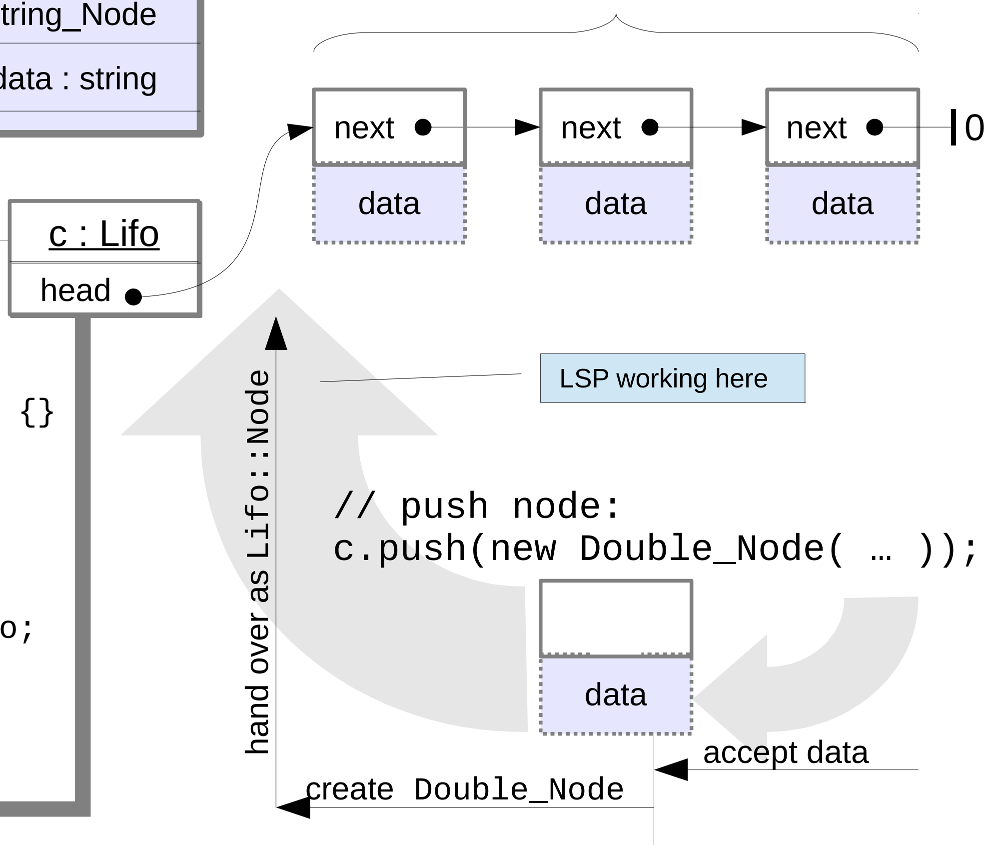
(Node-) Inheritance Technique

API based on pointers and thus implicitely polymorphic

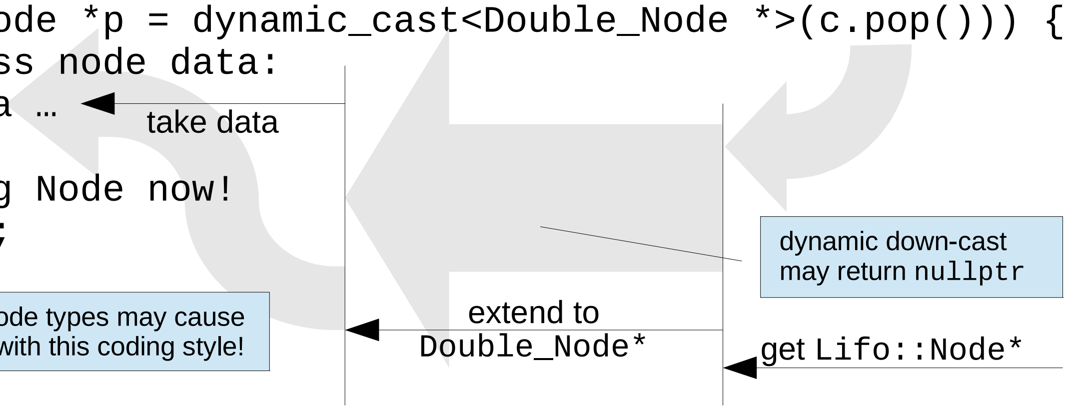


Stored Elements:
- formally `Lifo::Node`-s
- actually derived classes

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

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




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

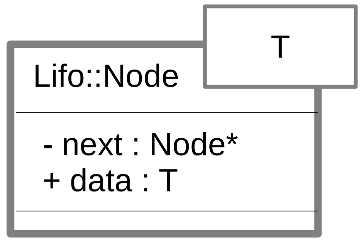



```
// zero run-time overhead (may cause undefined behavior):
… static_cast<Double_Node *>(p)->data …
```

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

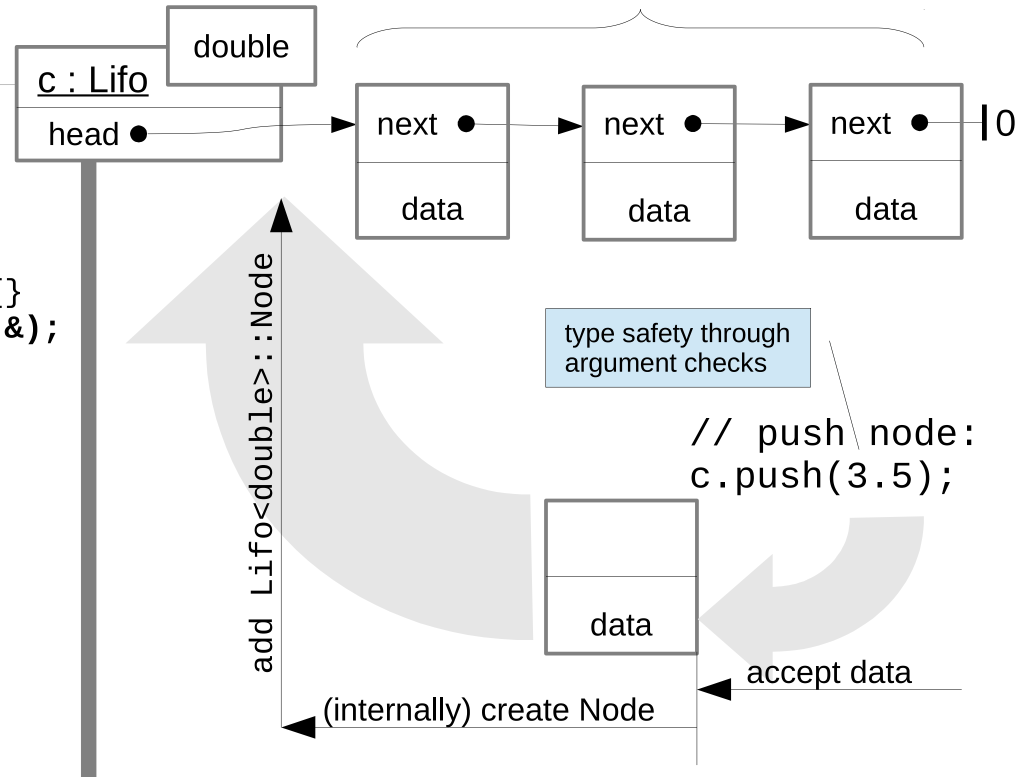
Template Technique

API based on values and thus non-polymorphic by default

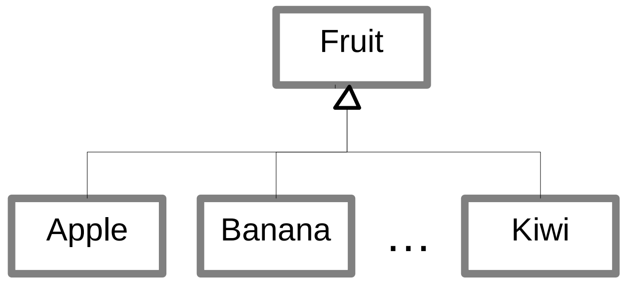


Stored Elements:
- `Lifo<double>::Node`-s

```
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> …
```




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



Polymorphic Elements

for polymorphic 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

```
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!

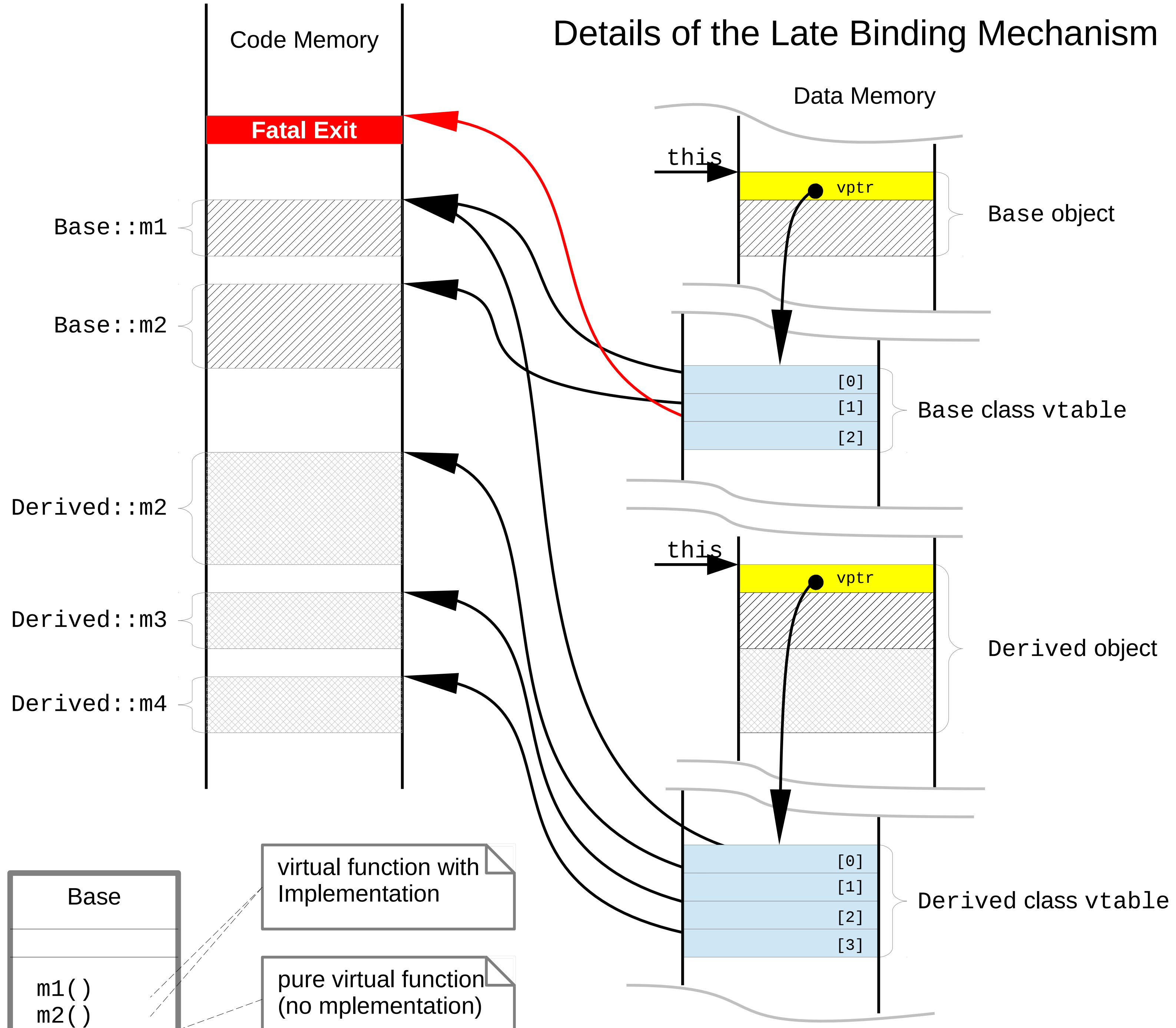
OK but, just a `Fruit` … (Sorry Sir, no Banana flavour today!)

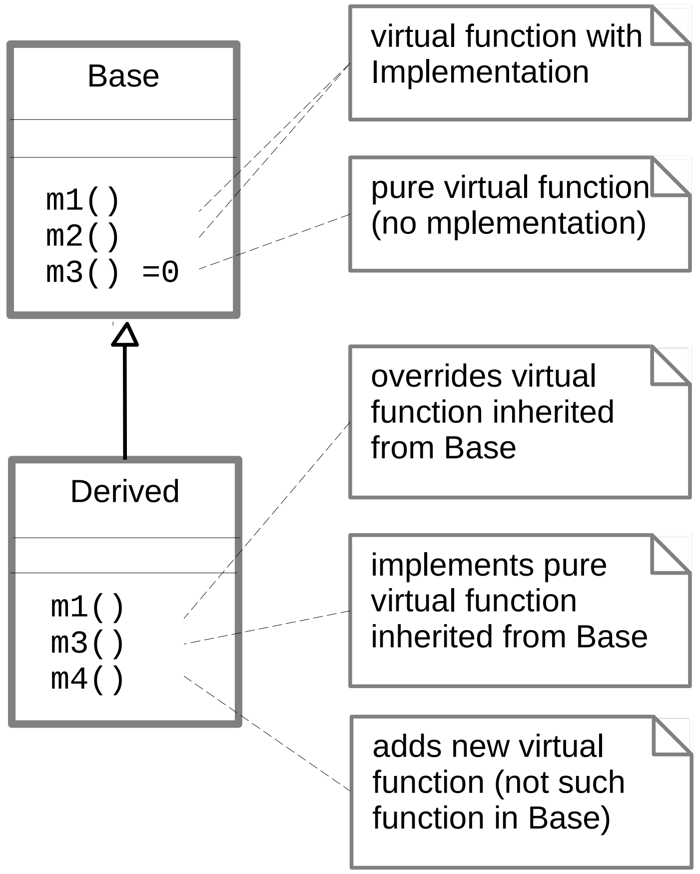
does not compile because all what is returned are `Fruit`-s

Centainer Implementation Techniques

Run-time Type Identification

Details of the Late Binding Mechanism





Virtual function call in detail:

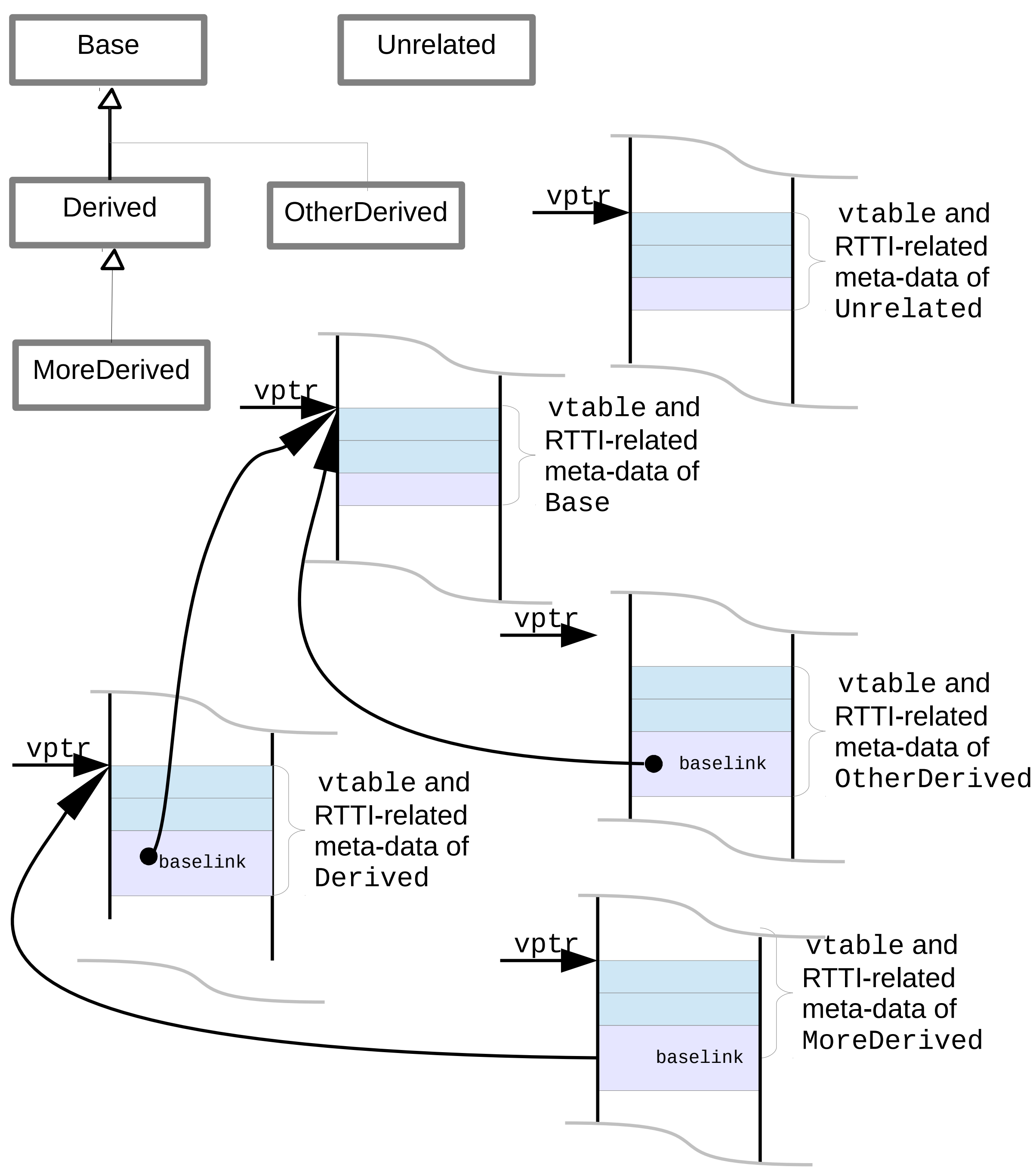
- starting from the `this` pointer
- locate the `vtable`[1] via the `vptr`[2]
- add member function specifiy offset
- read address from code segment
- branch to this location as sub-routine

–

[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 classes with at least one virtual member function:

- This avoids overhead which would otherwise occur on per object.
- Meta-data is stored in the vincinity 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).

Refactoring RTTI …

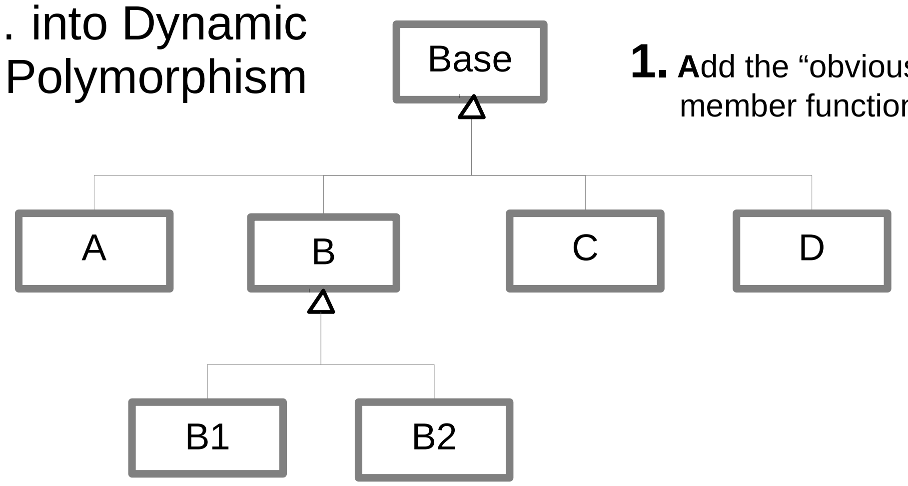
```
void foo(Base &r) {
    …
    if (auto p = dynamic_cast<A*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<B1*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<B2*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<C*>(&r)) {
        …
    }
    if (auto p = dynamic_cast<D*>(&r)) {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

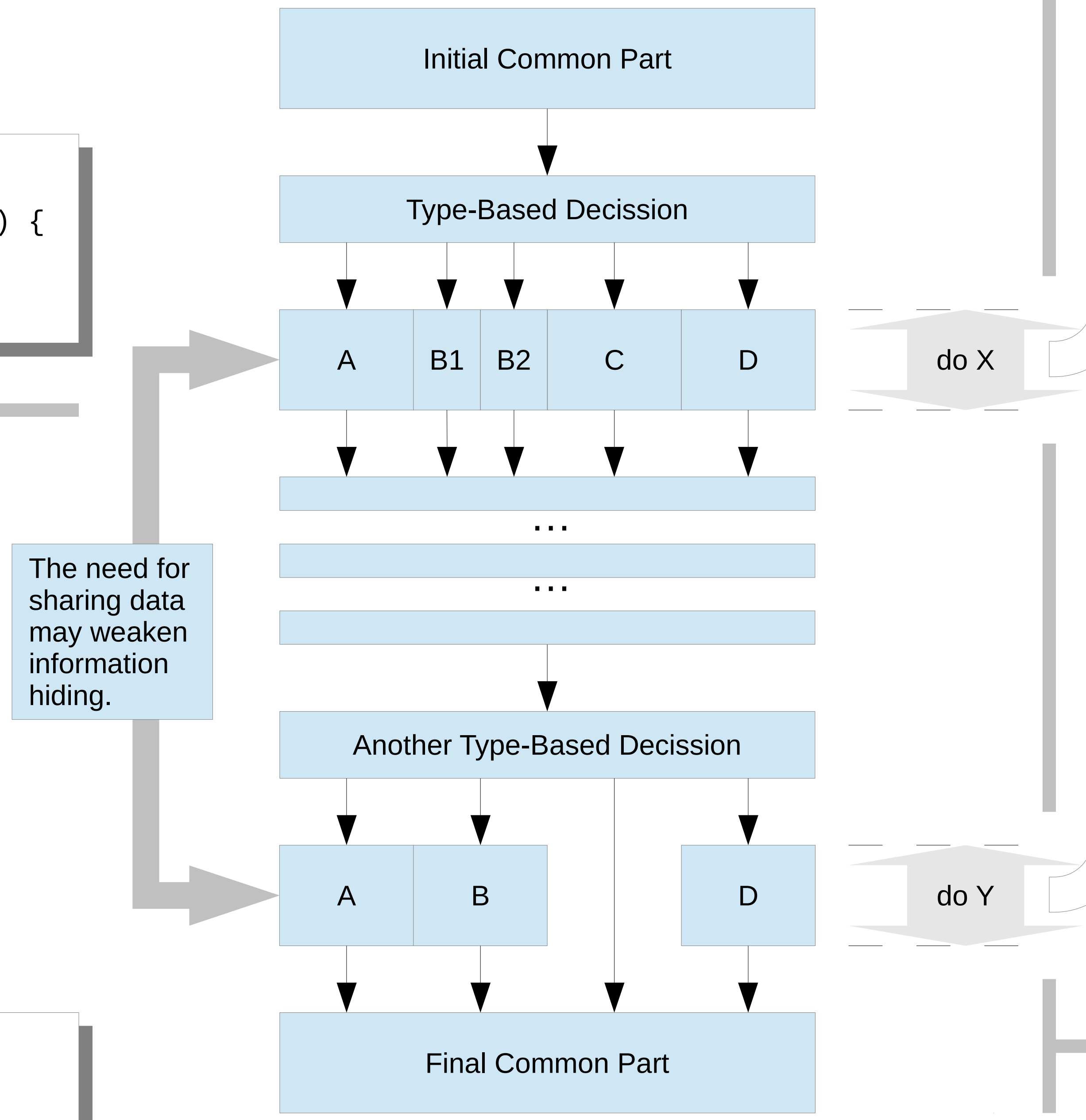
```
    …
    // combining B1 and B2
    if (auto p = dynamic_cast<B*>(&r)) {
        …
    }
    …
```

```
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)) {
        …
    }
    …
}
```

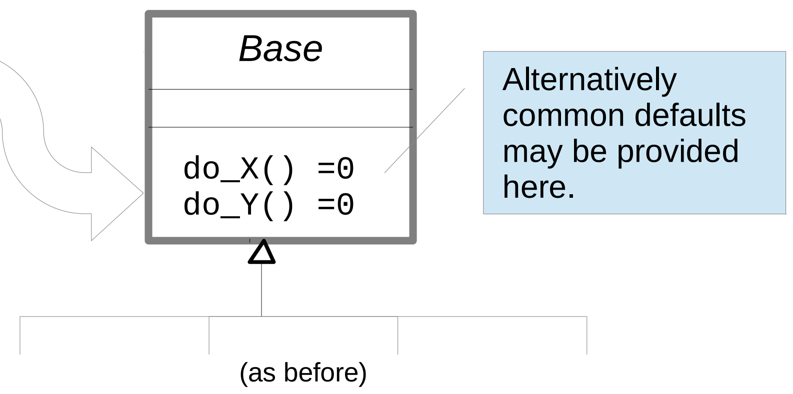
```
    …
    // combining B1 and B2
    if (typeid(r) == typeid(B1)
     || typeid(r) == typeid(B2)) {
        …
    }
    …
```

… into 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() {
    …
}
```

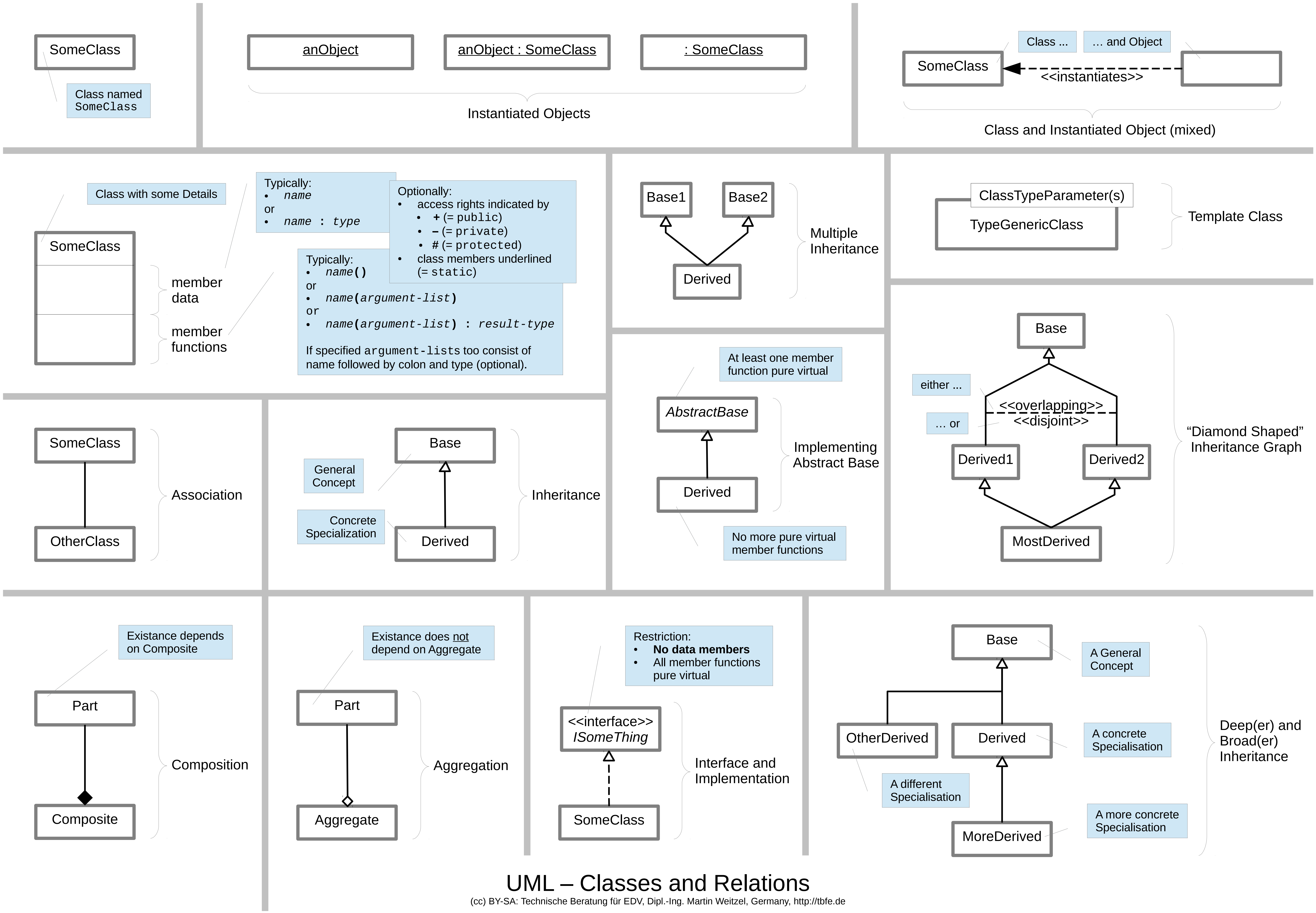
```
void A::do_Y() {
    …
}
void B::do_Y() {
    …
}
void C:: do_Y() {
    /*empty*/
}
void D:: do_Y() {
    …
}
```

Data can be shared easily in privacy.

3. Replace multiway branches with member function calls:

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

Type-Based Multiway Branching



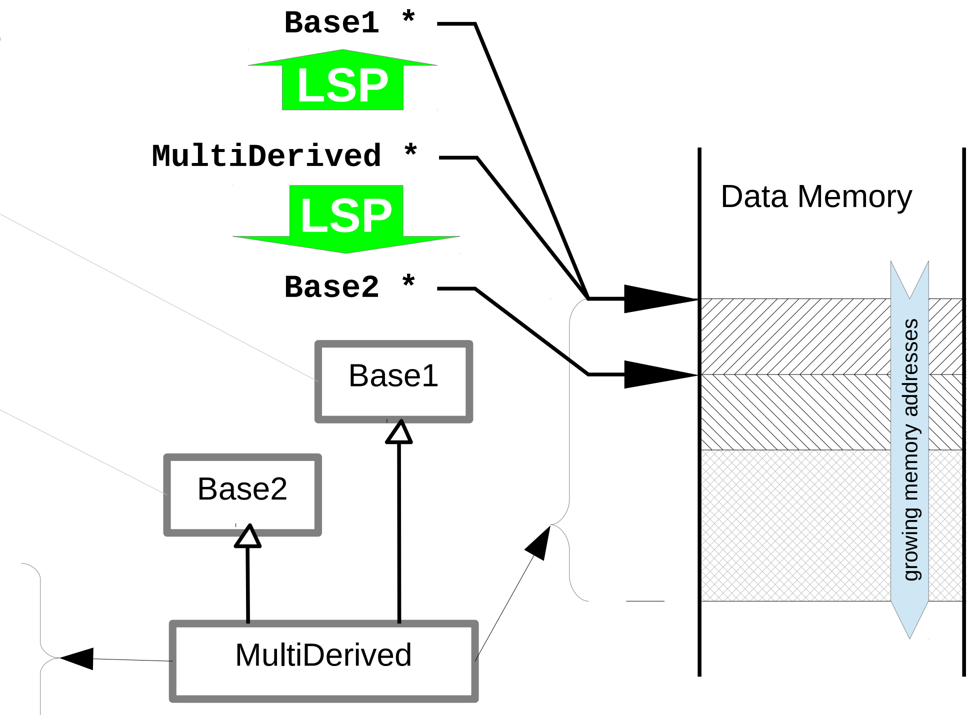
UML – Classes and Relations

Multiple Inheritance

```
class Base1 {

};
```

```
class Base2 {

};
```

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

};
```

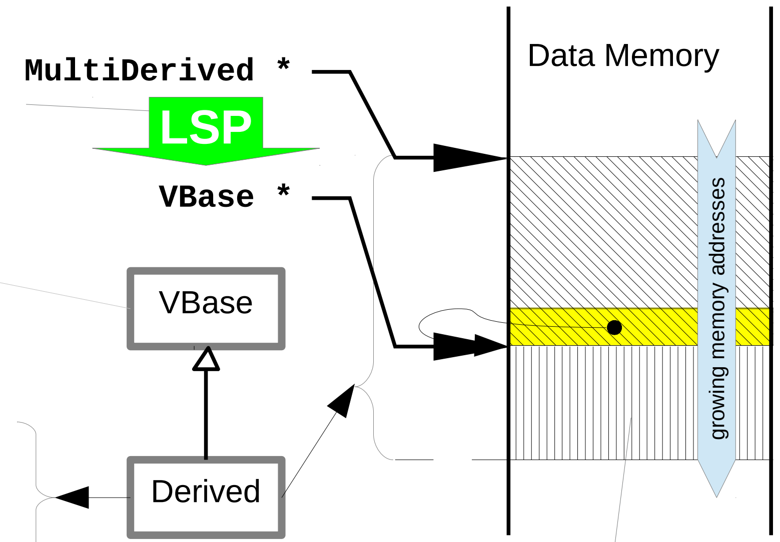


Virtual Base Class

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

```
class VBase {

};
```

```
class Derived
    : virtual public VBase {

};
```



Memory layout is the same for `virtual` **`Brivate`** 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.

Overlapping Common Base Class

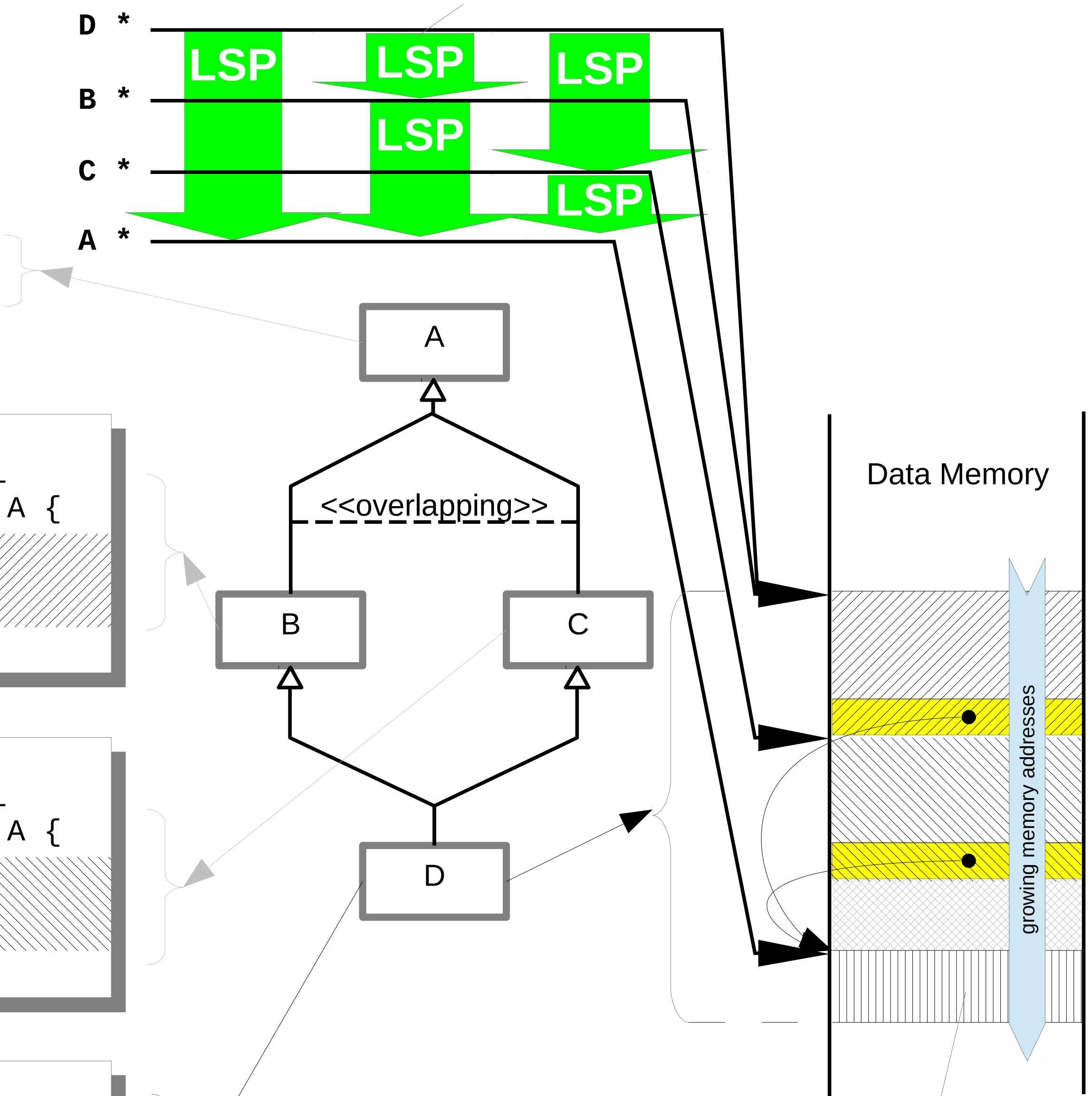
No LSP for `virtual` **`private`** base classes.

```
class A {

};
```

```
class B
    : virtual
      public A {

};
```

```
class C
    : virtual
      public A {

};
```

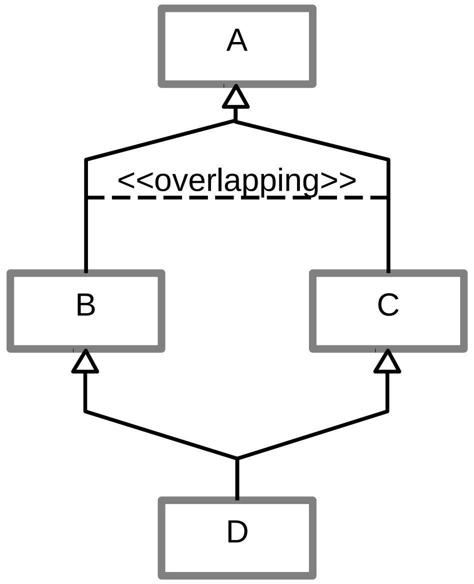
```
class D
    : public B,
      public C {

};
```



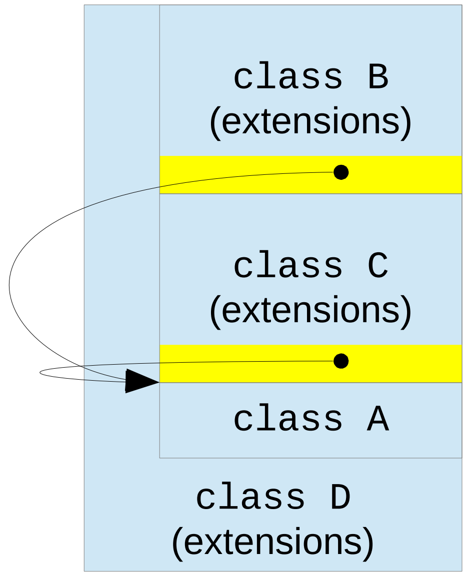
Memory layout is the same for `virtual` **`private`** base classes.

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



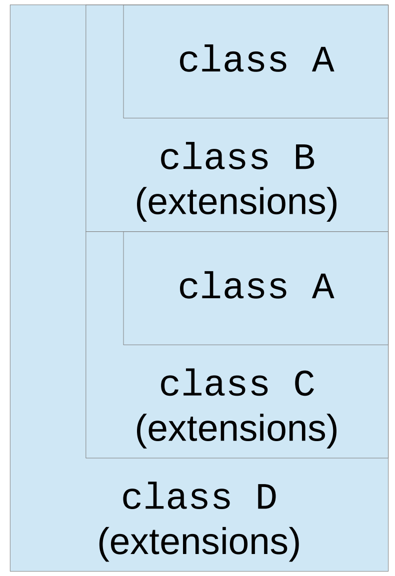
UML Class Graph



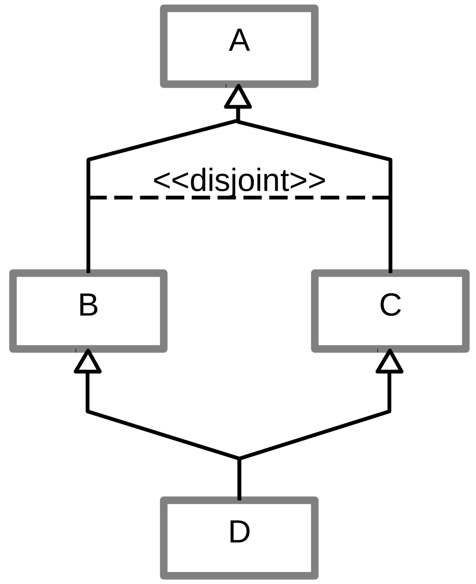
Member Data to Memory Mapping

Up-Casts by LSP

Automatic Type Conversions		
to ←	*from*	*→ to*
A	A	A
A	B	A
A	C	A
A, B, C	D	B, C



Member Data to Memory Mapping



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( … )
{ … };
```

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()`	

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.**

C++ Source

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

```
A::A( … ) { … };
```

```
A::A( … ) { … };
```

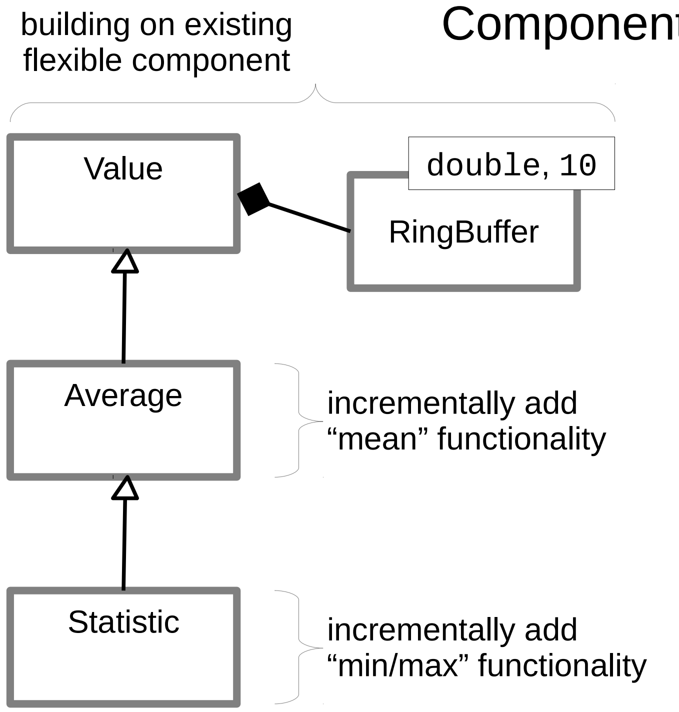
```
B::B( … )
    : A( … )
{ … };
```

```
C::C( … )
    : A( … )
{ … };
```

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

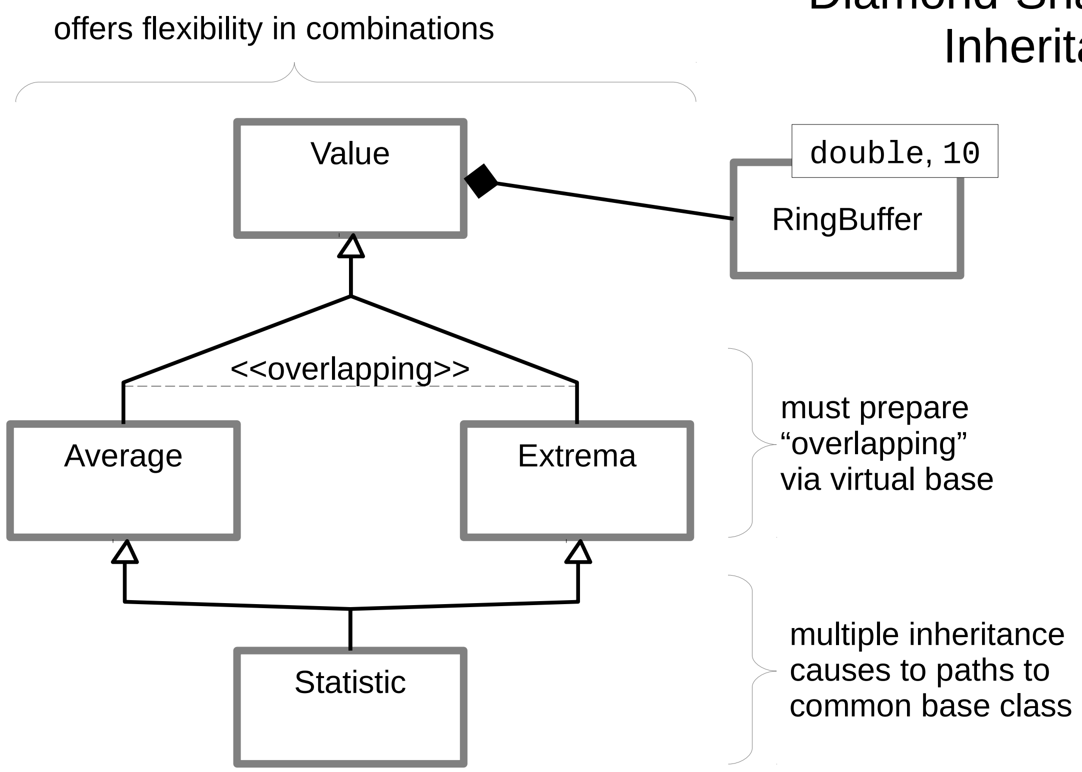
Diamond Shaped Inheritance

Reusing Adapted Component



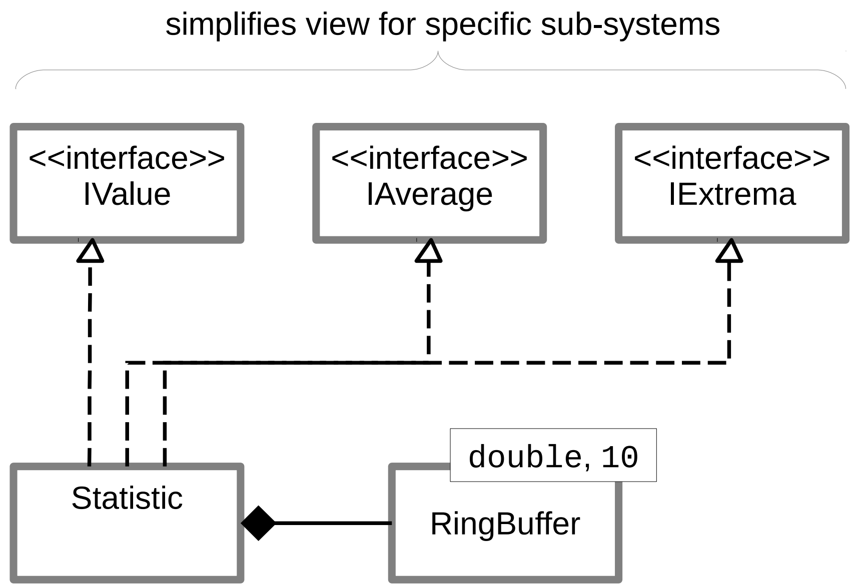
Simple design using:
- template (*Instantiation of RingBuffer*)
- composition (*Value has a RingBuffer*)
- base clases (*Average is a Vaue* and *Statistic is a Average*)

Diamond-Shaped Inheritance

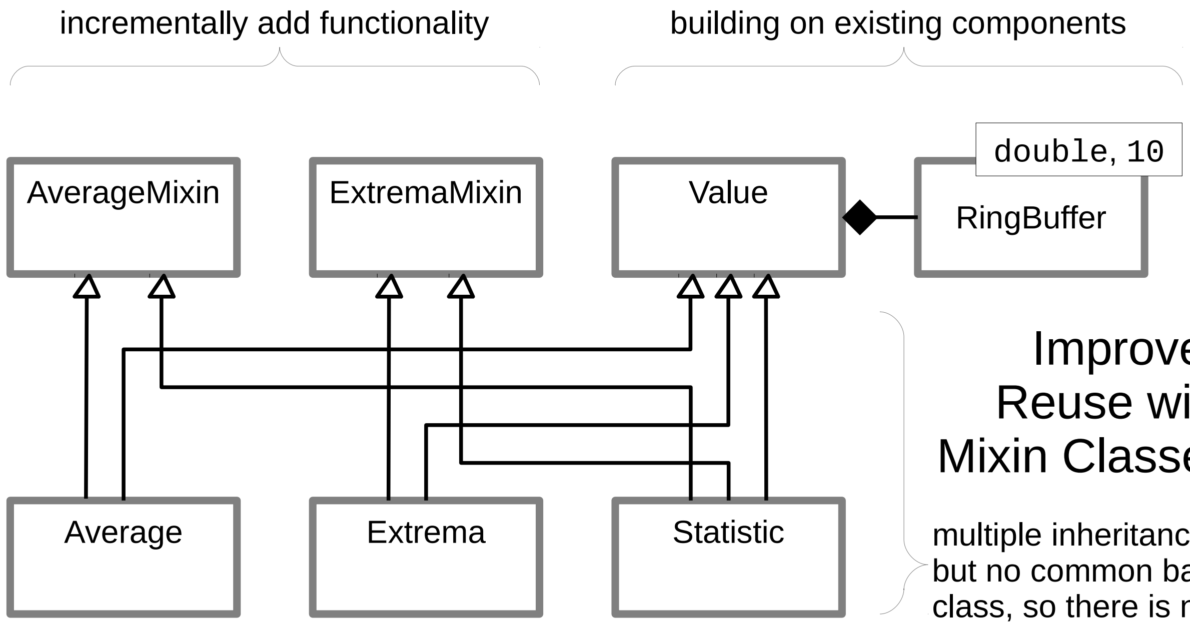


More flexible design with “diamond shaped” inheritance:
- each of the classes (*Value*, *Average*, *Extrema*, *Statistic*) may be used on its own
- intermediate classes (*Average*, *Value*) have to pay the “price” …
- … for simple re-use in the most derived class (*Statistic*)

Three Interfaces



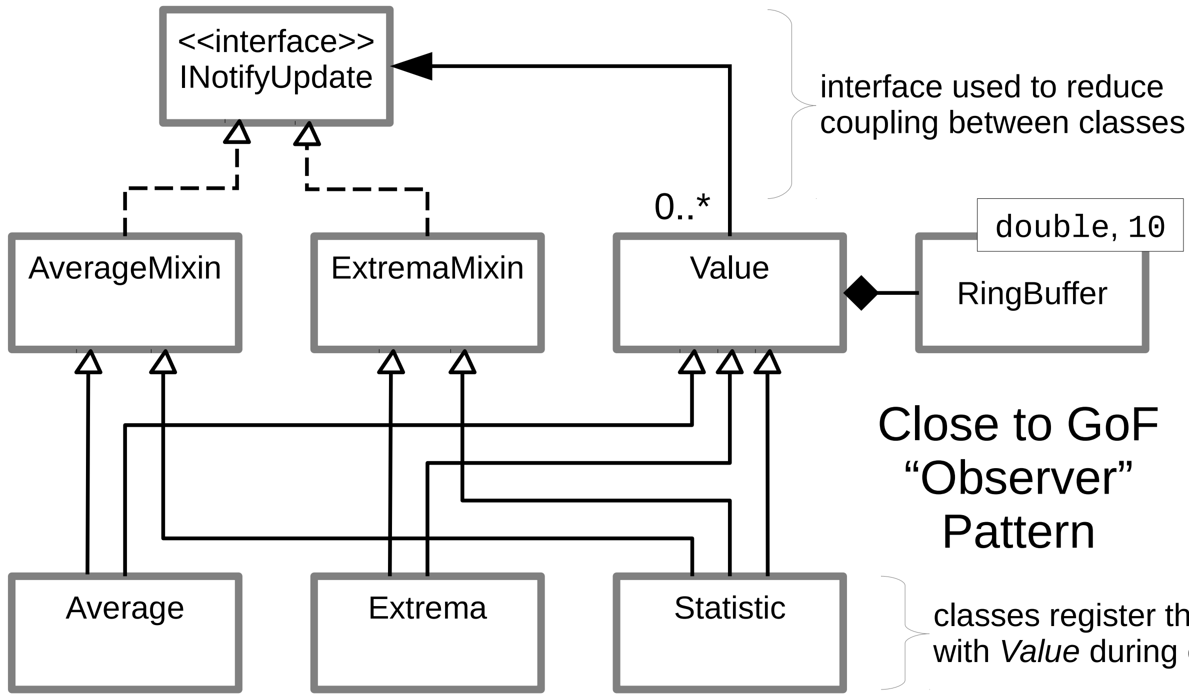
Alternative design with interfaces hides complexity from clients that do not need to know details:
- some clients may only need to handle Values (→ to know *IValue* is sufficient)
- others may only need to handle Averages (→ to know *IAverage* is sufficient)
- Yetl others may only need to handle Extrema (→ to know *IExtrema* is sufficient)



Improved Reuse with Mixin Classes

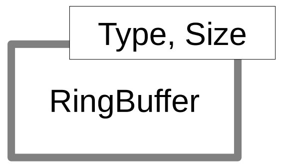
multiple inheritance but no common base class, so there is no need for virtual base

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*



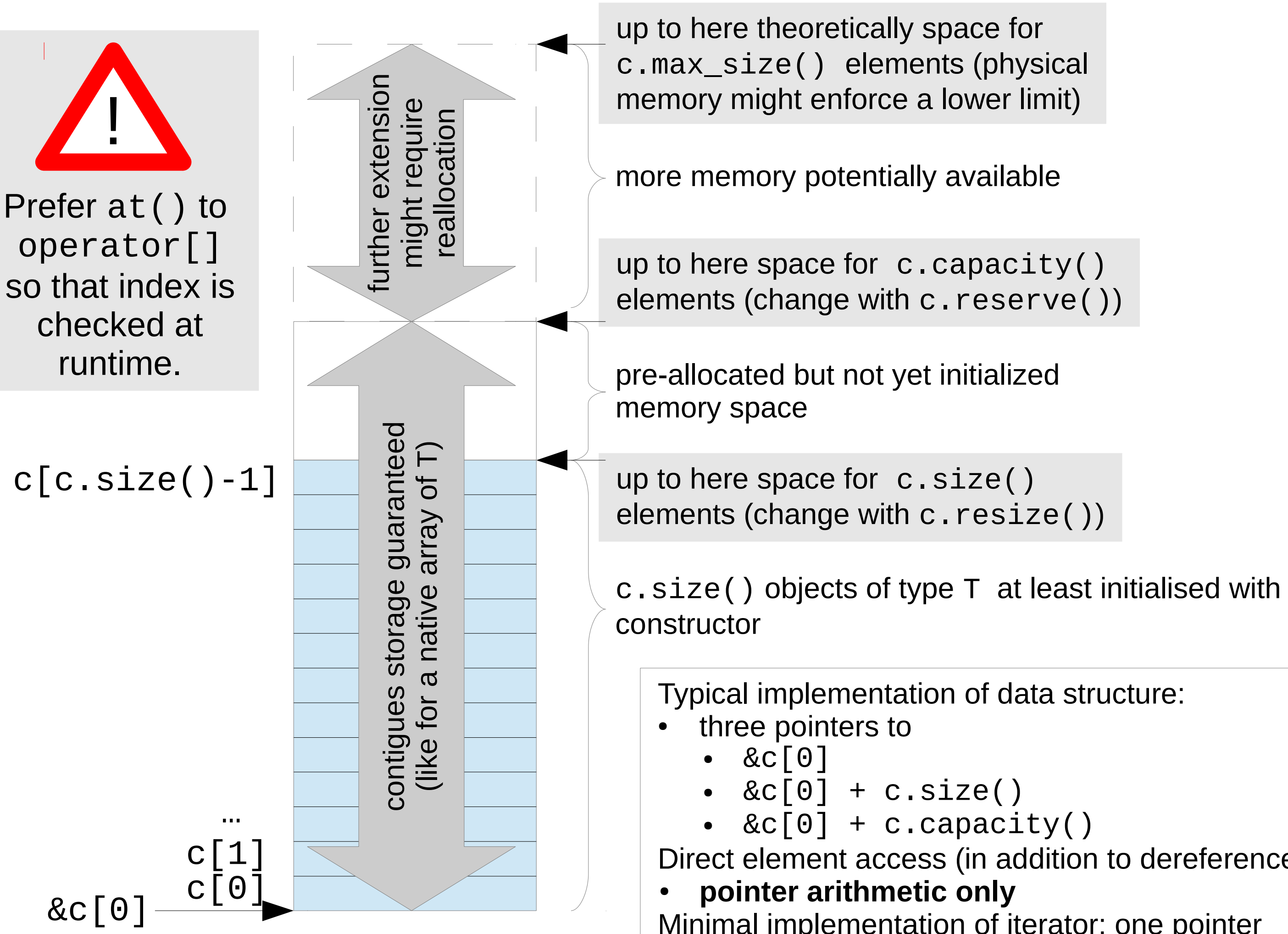
Reusable Component

Generic class parametrized in
- **type** and
- **number**

of elements

Examples – Classes and Relations

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

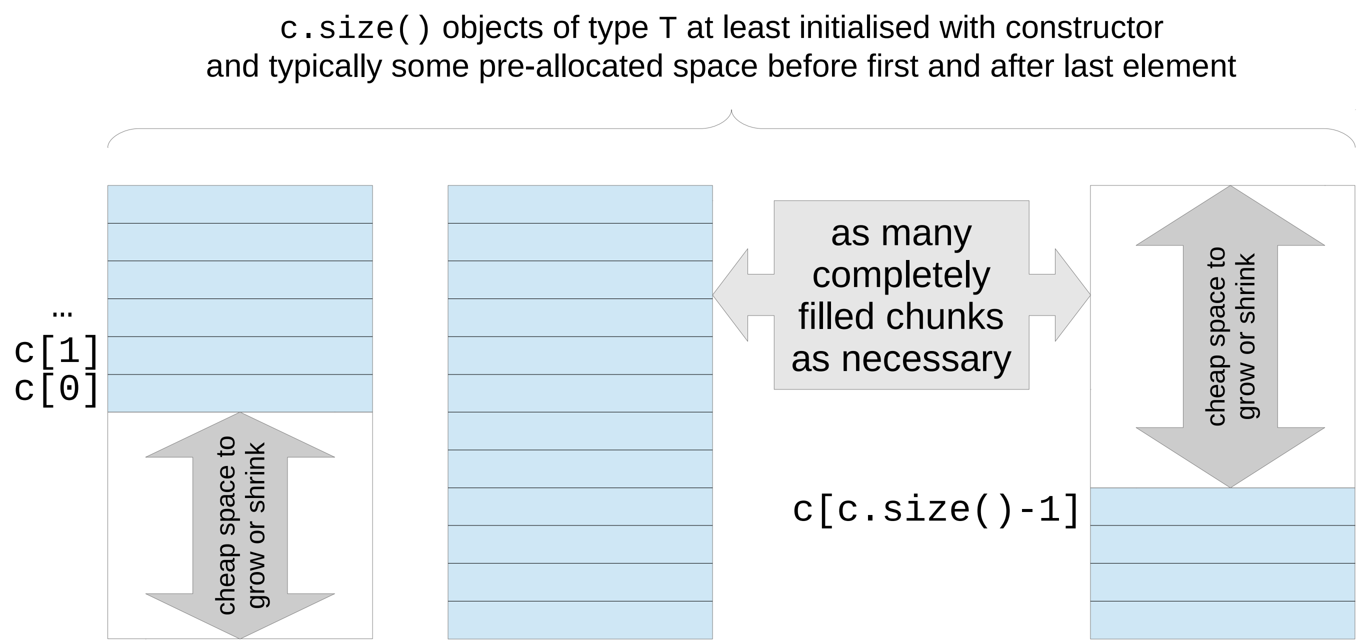


Typical implementation of data structure:
- three pointers to
  - `&c[0]`
  - `&c[0] + c.size()`
  - `&c[0] + c.capacity()`

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

`std::vector<T> c;`



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

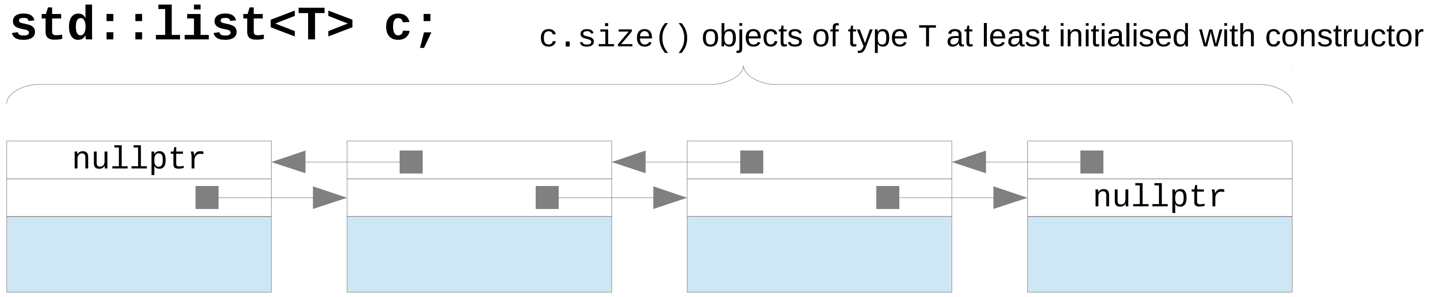
Typical implementation of data structure:
- pointer to first and last element
- one more pointer to
  - **additional block holding pointers to chunks**
- integral value for number of elements

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

`std::deque<T> c;`

`std::list<T> c;`

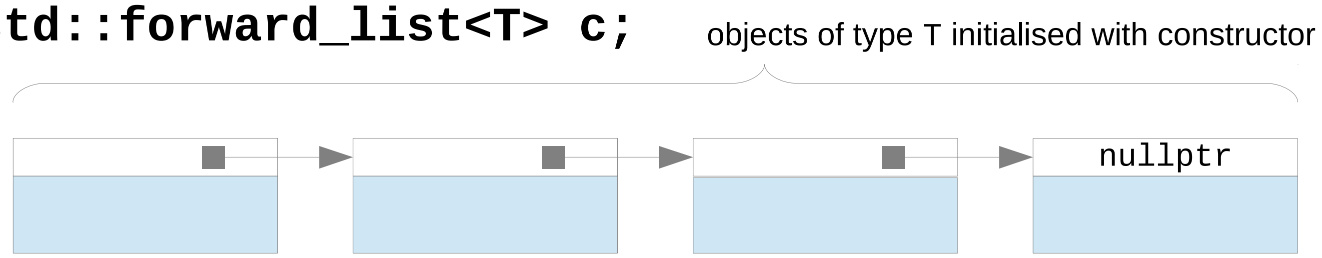


Typical implementation:
- **two pointers per element**
- pointer to first and last element
- integral value for number of elements

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.

`std::forward_list<T> c;`

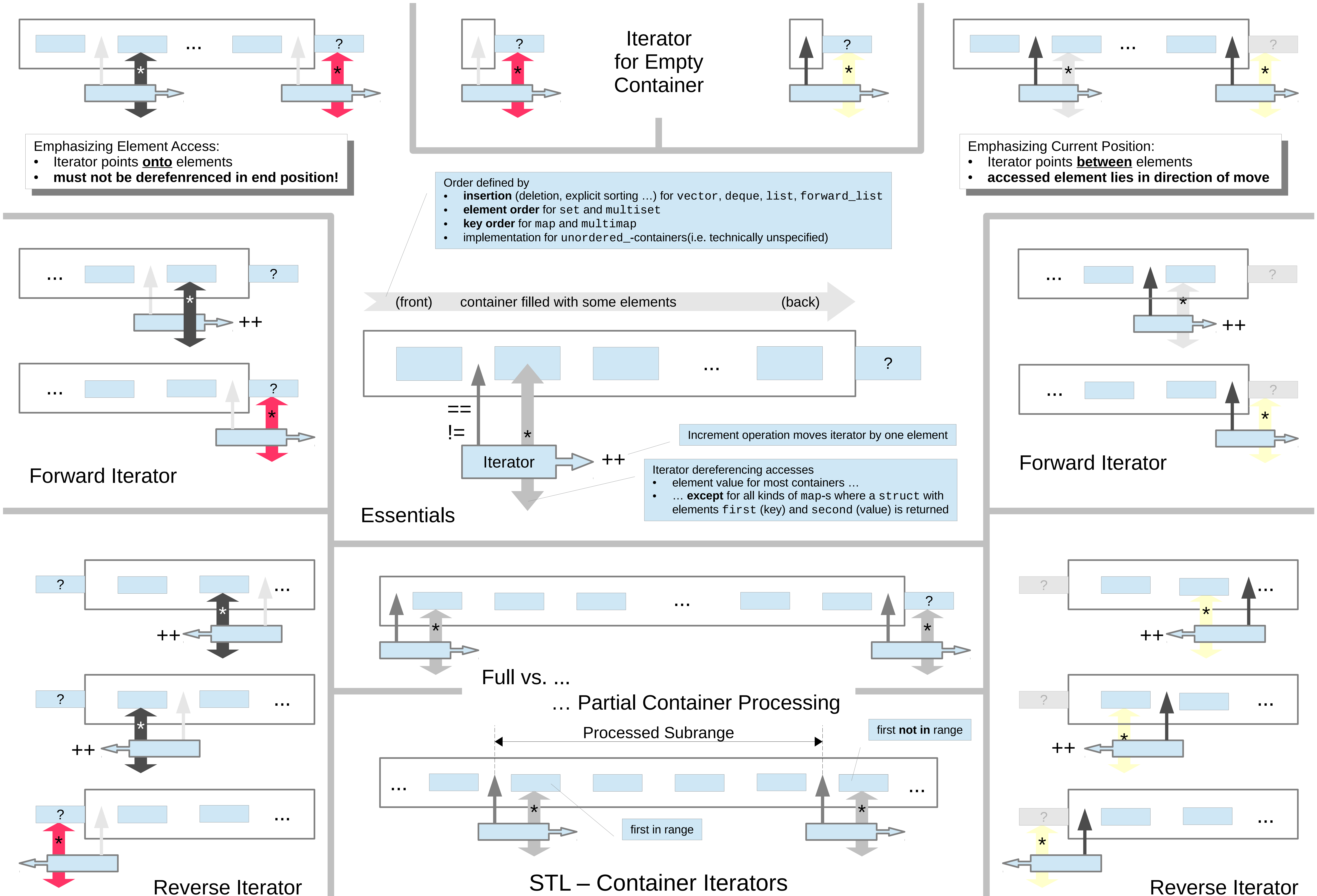


Use `c.empty()` to check wether elements exist.

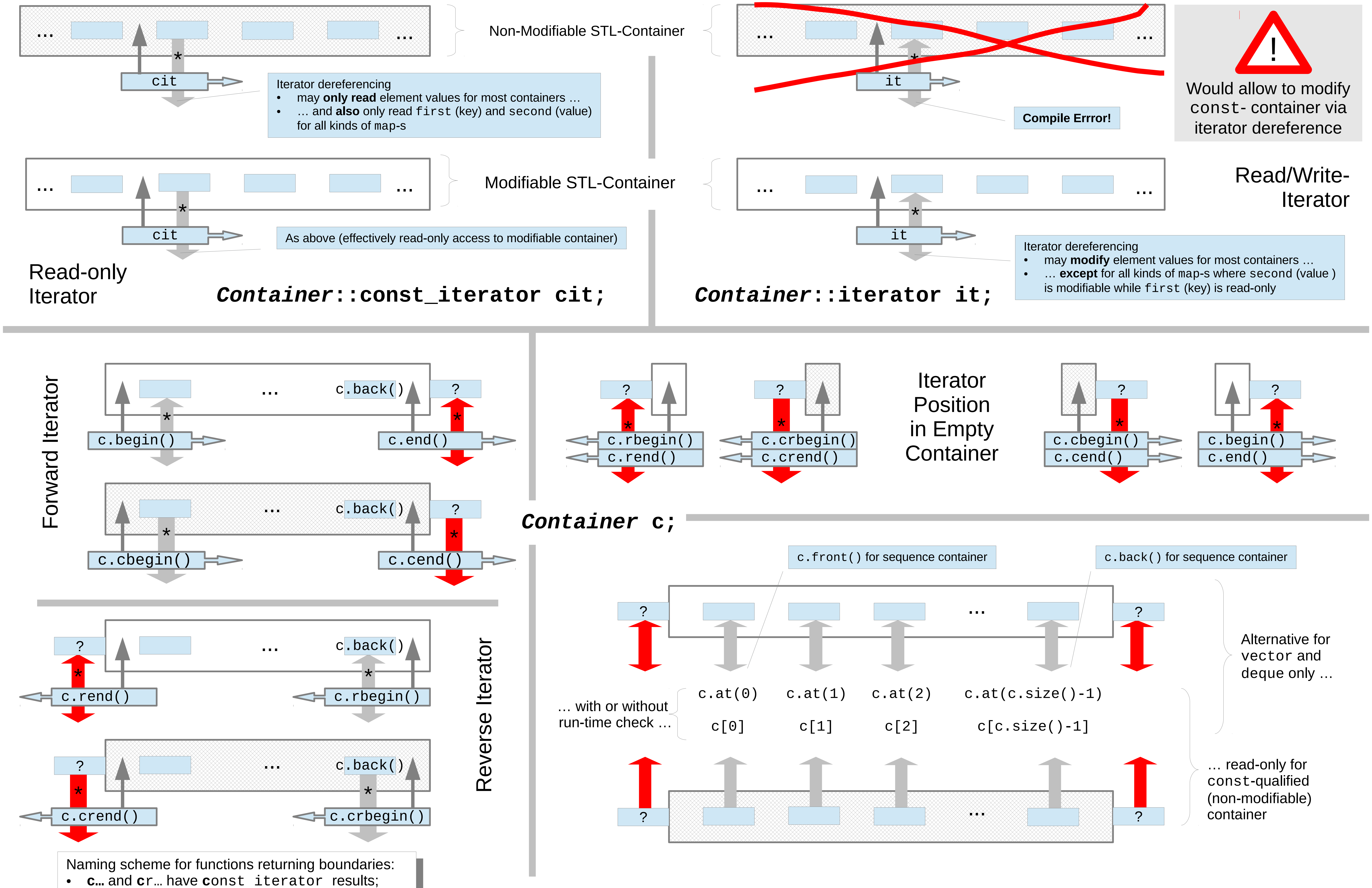
Typical implementation:
- **one pointer per element**
- only pointer to first element
- **number of elements not stored!**

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

STL – Sequence Container Classes



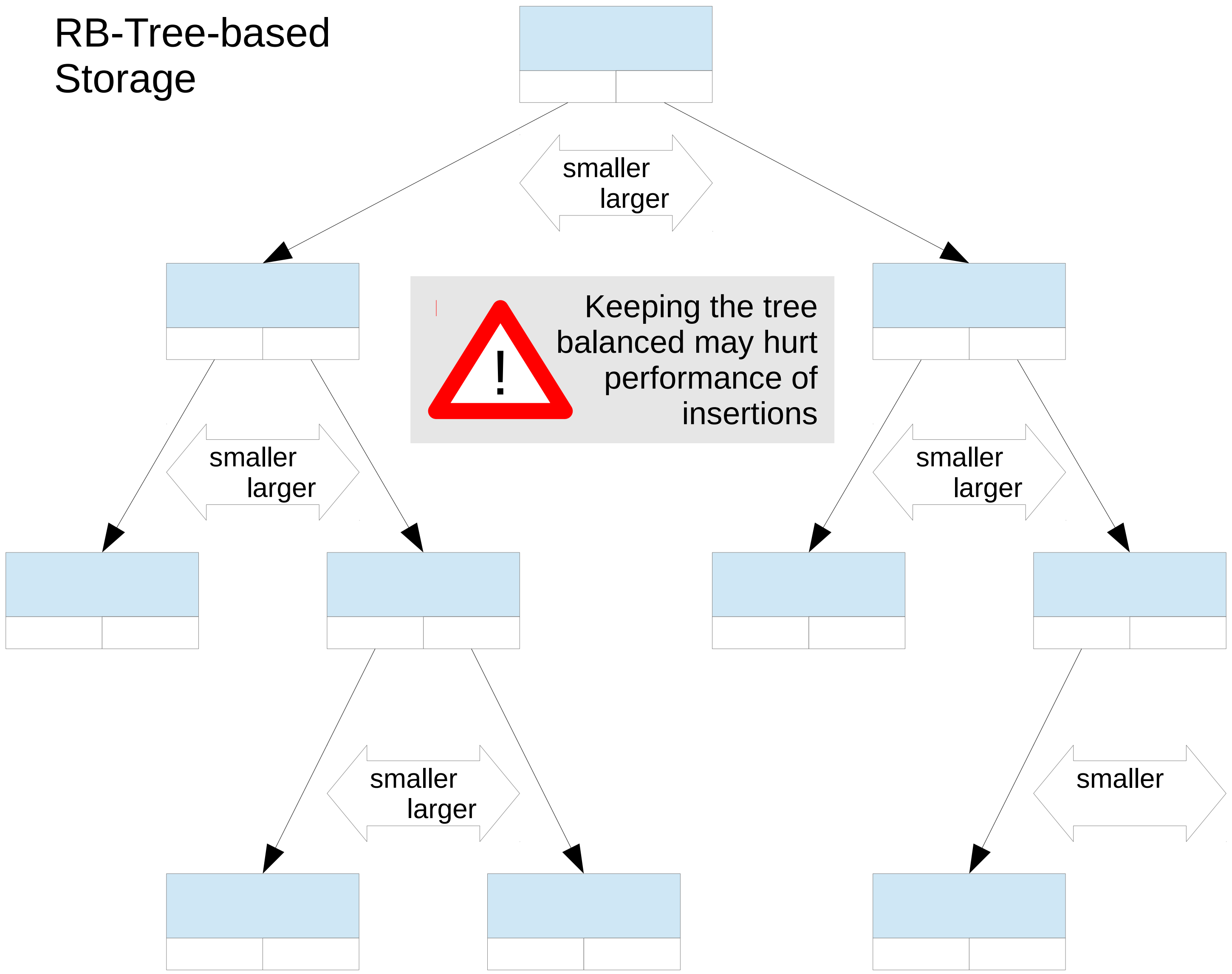
STL – Container Iterators

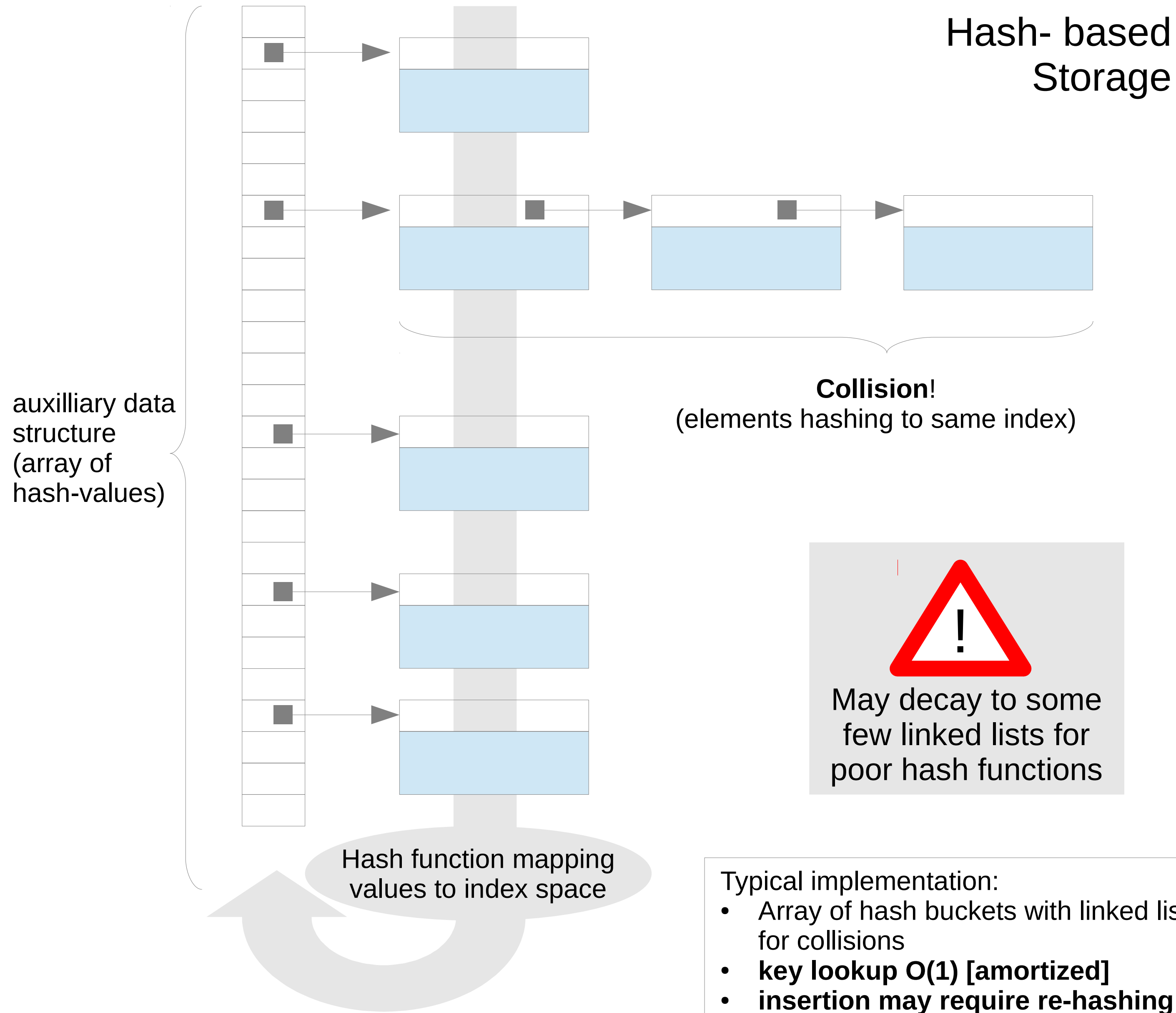


Accessing Element via Index

STL – Iterator Details

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)



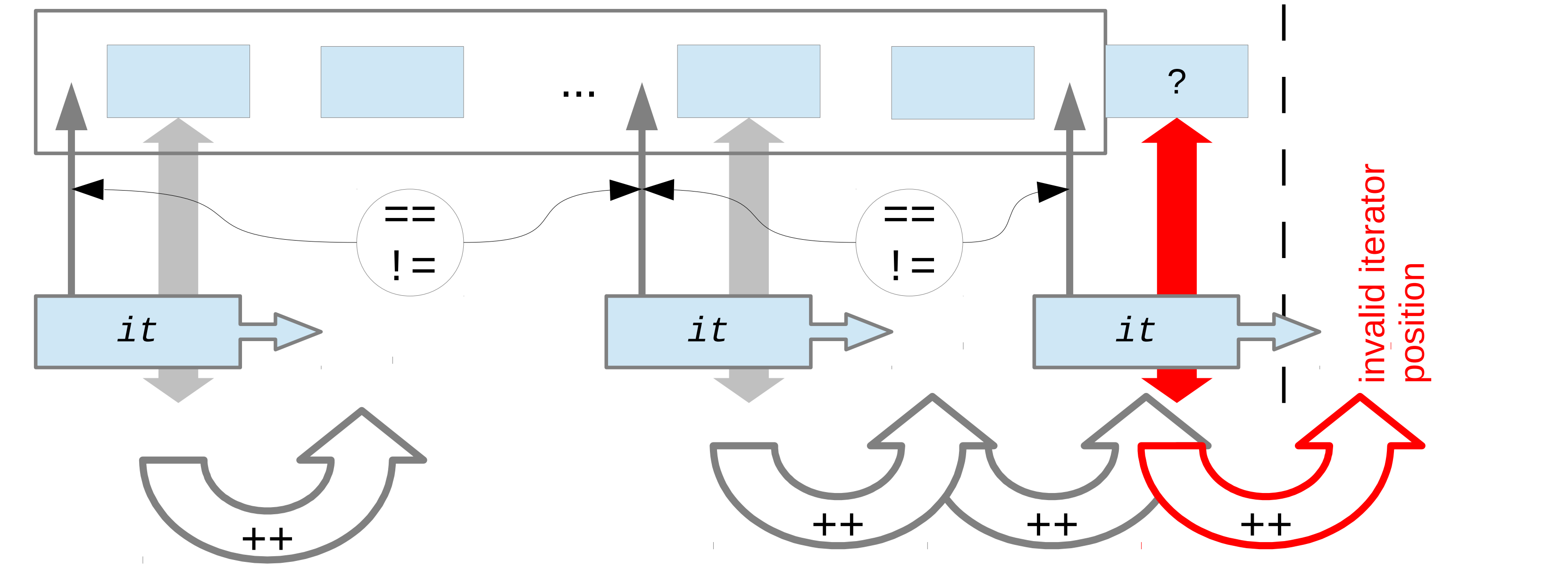


Typical implementation: Black-Red-Tree
- **key lookup $O(\log_2 N)$**
- **insertion may require re-balancing**
- two pointers per element

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.

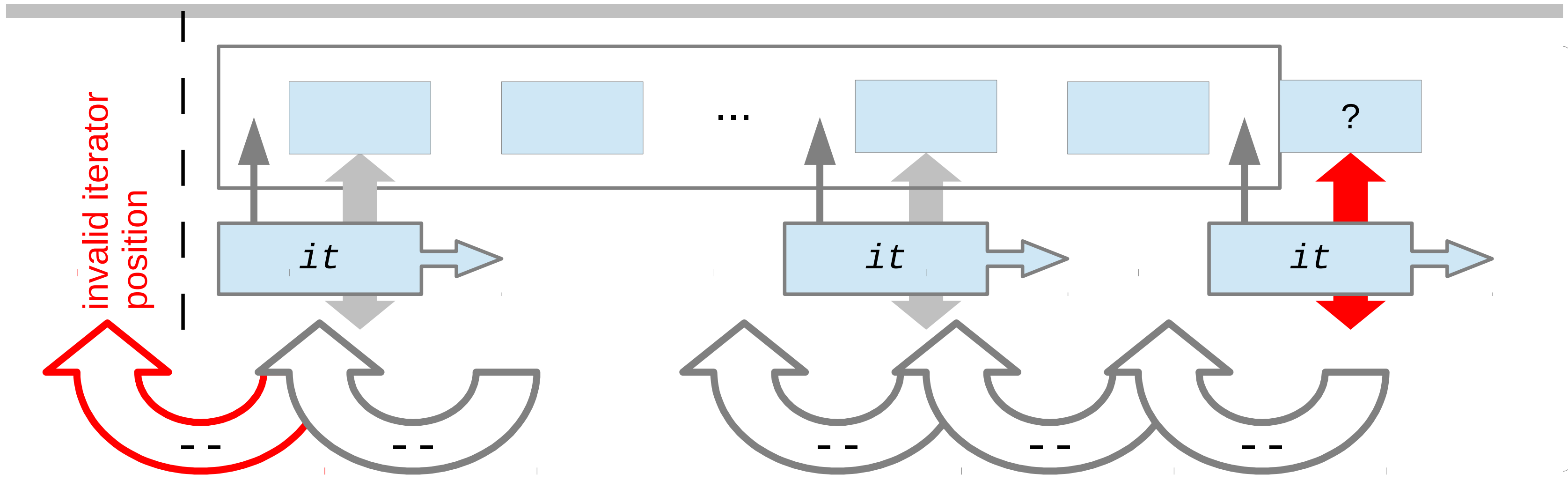
Typical implementation:
- Array of hash buckets with linked lists for collisions
- **key lookup O(1) [amortized]**
- **insertion may require re-hashing**
- one pointer per element
- for good performance ~20% oversized array of pointers for maximum number of elements

STL – Associative Container Classes



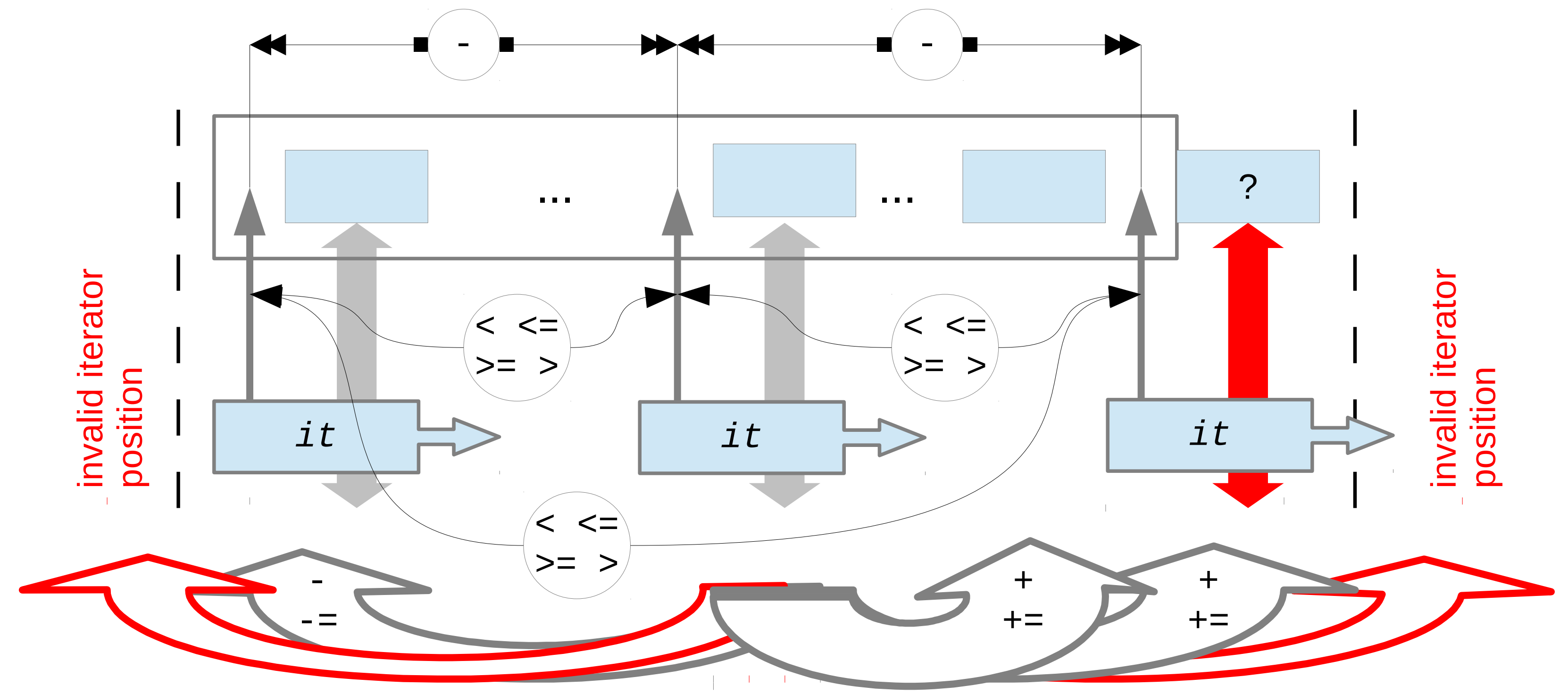
Operations of **Unidirectional Iterators**

	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**

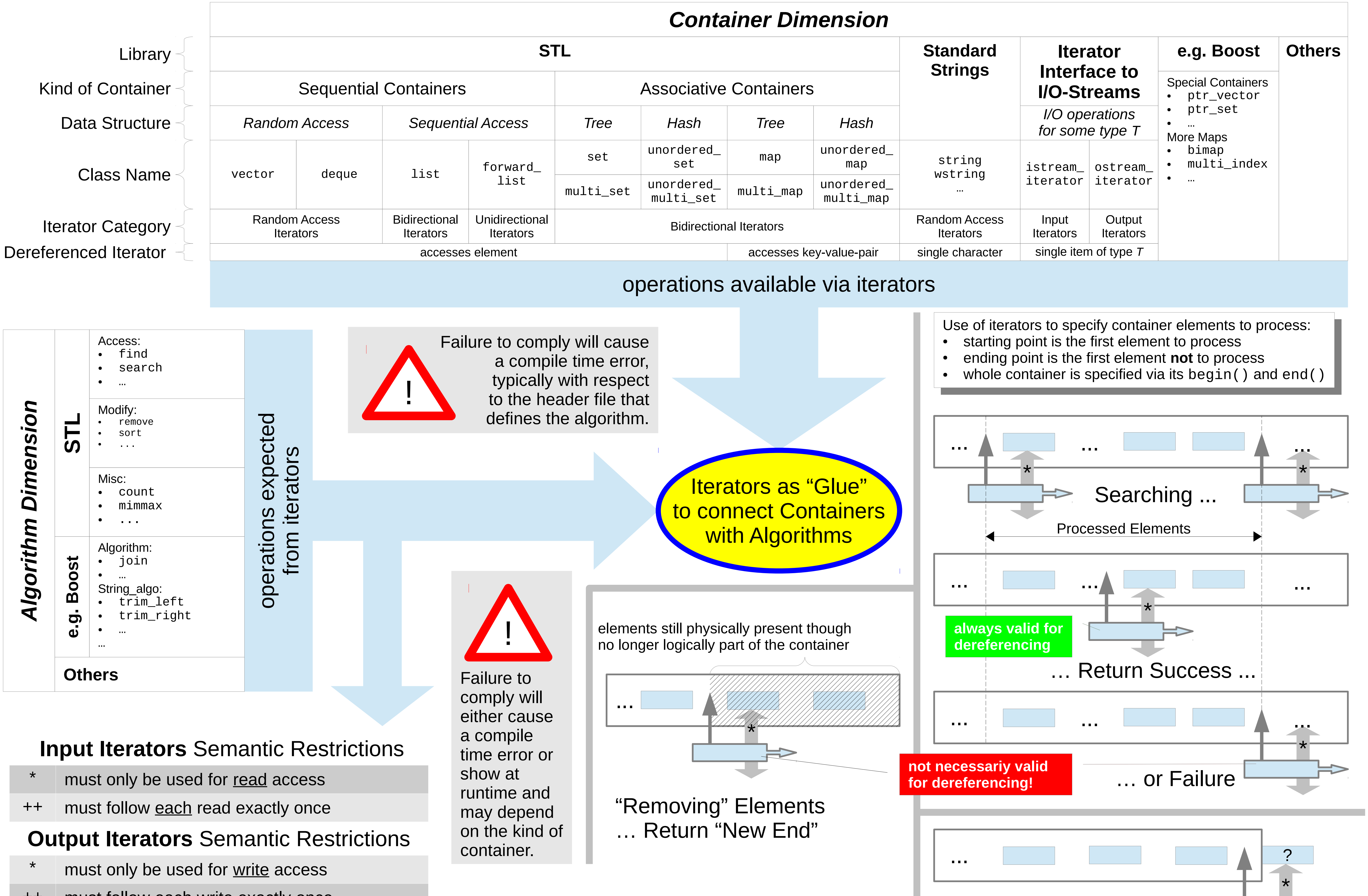
	Effect	*Remarks*
`--it` `it--`	advance to previous element (usual semantic for pre-/postfix version)	**undefined at container begin**



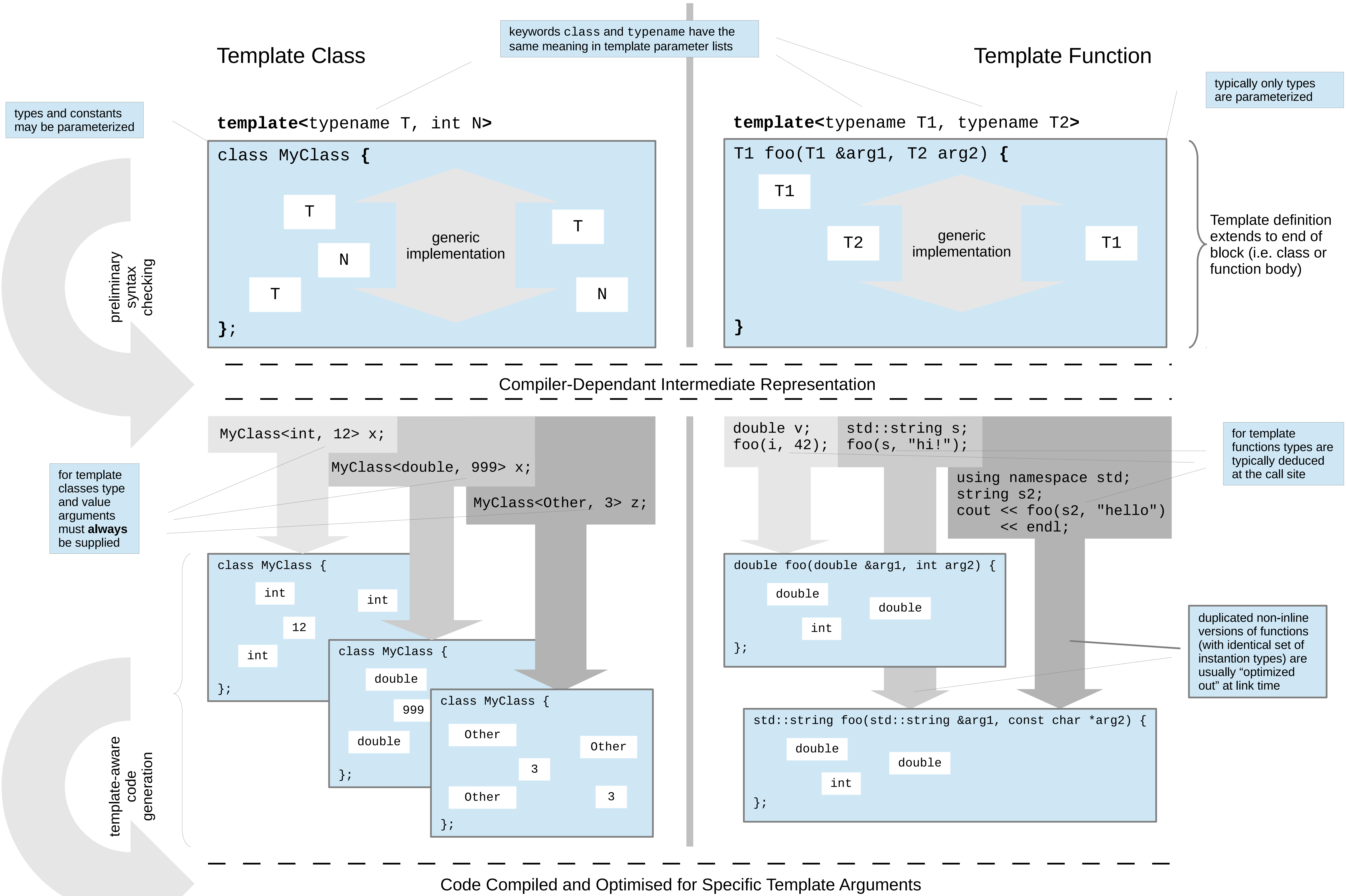
Additional Operations of **Random Access Iterators**

	Effect	*Remarks*
`it + n` `it += n`	`it` advanced by *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 by *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`	

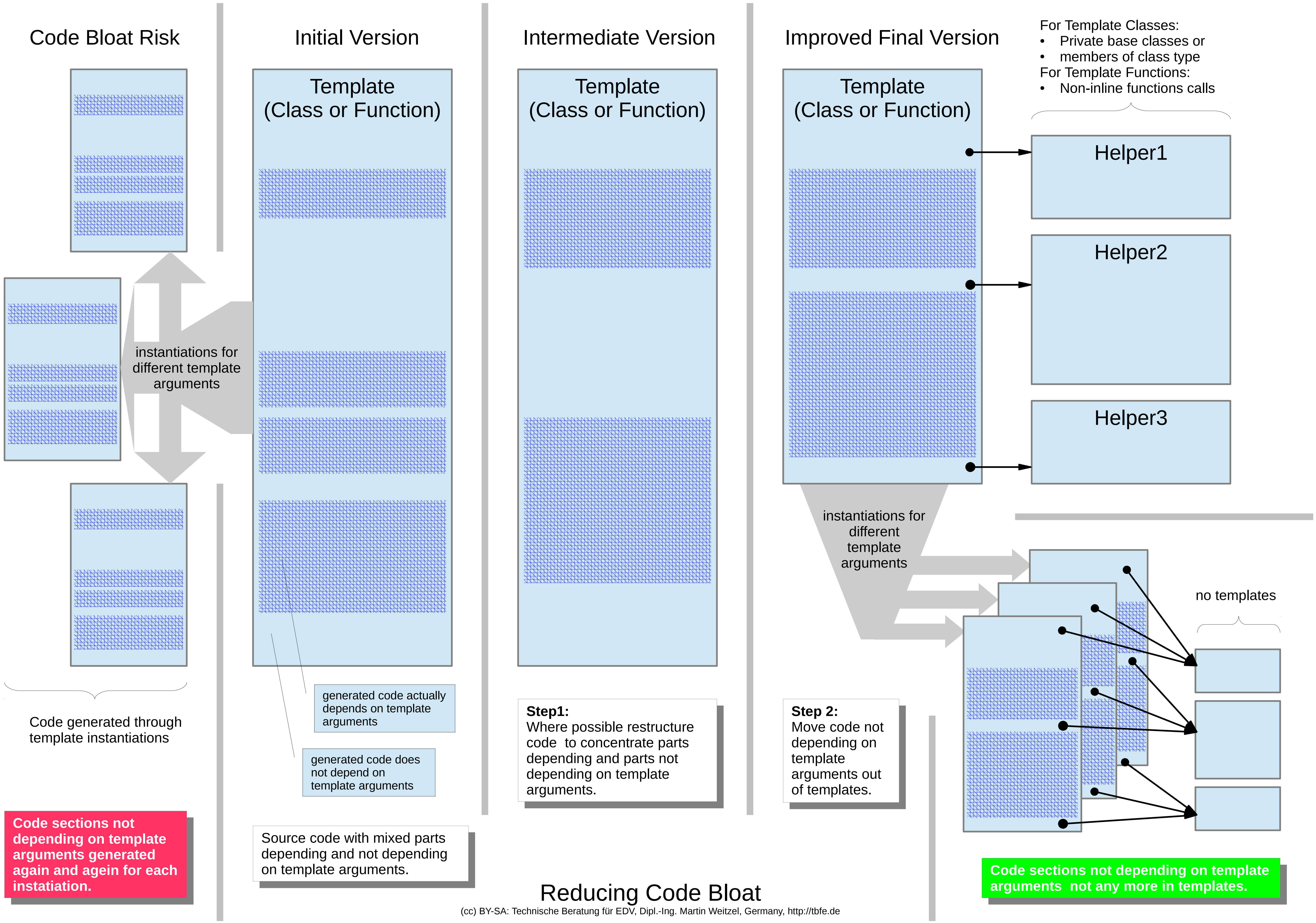
STL – Iterator Categories

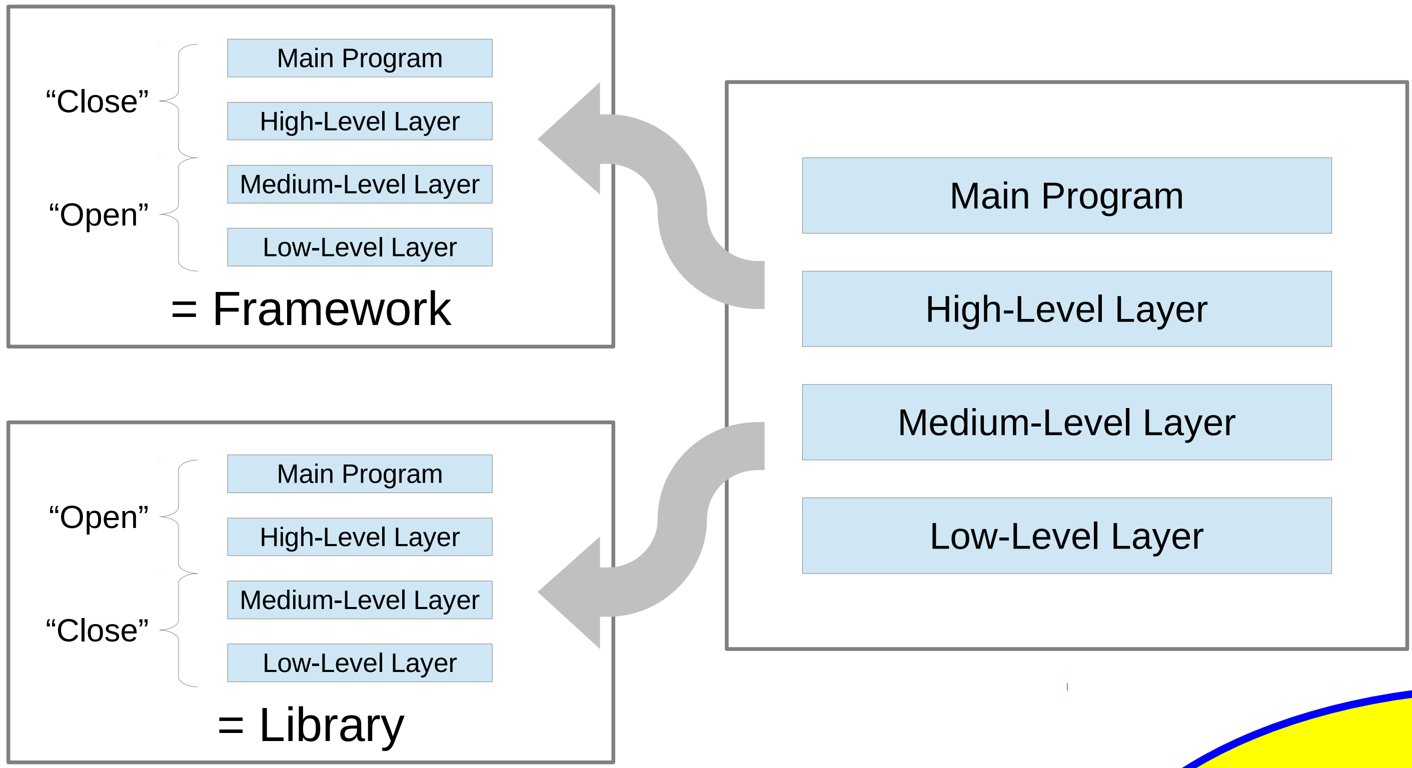


STL – Iterator Usages



Template Basics


Code Bloat Risk
Initial Version
Intermediate Version
Improved Final Version
For Template Classes:
• Private base classes or
• members of class type
For Template Functions:
• Non-inline functions calls
Template
(Class or Function)
Template
(Class or Function)
Template
(Class or Function)
Helper1
Helper2
Helper3
instantiations for different template arguments
generated code actually depends on template arguments
generated code does not depend on template arguments
Code generated through template instantiations
Step1:
Where possible restructure code to concentrate parts depending and parts not depending on template arguments.
Step 2:
Move code not depending on template arguments out of templates.
instantiations for different template arguments
no templates
Code sections not depending on template arguments generated again and agein for each instatiation.
Source code with mixed parts depending and not depending on template arguments.
Code sections not depending on template arguments not any more in templates.
Reducing Code Bloat
(cc) BY-SA: Technische Beratung für EDV, Dipl.-Ing. Martin Weitzel, Germany, http://tbfe.de

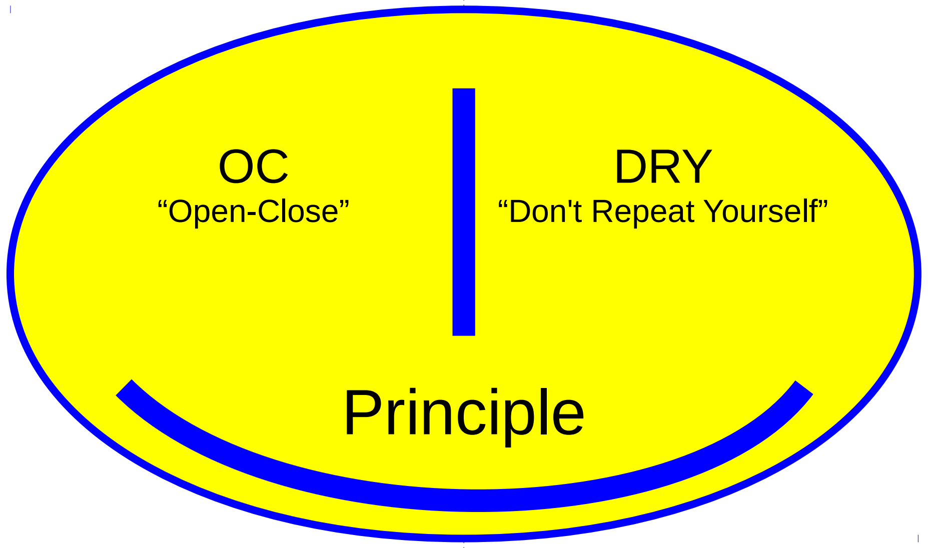


Design for Reusability:

- *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"?)

Parameterize 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.:

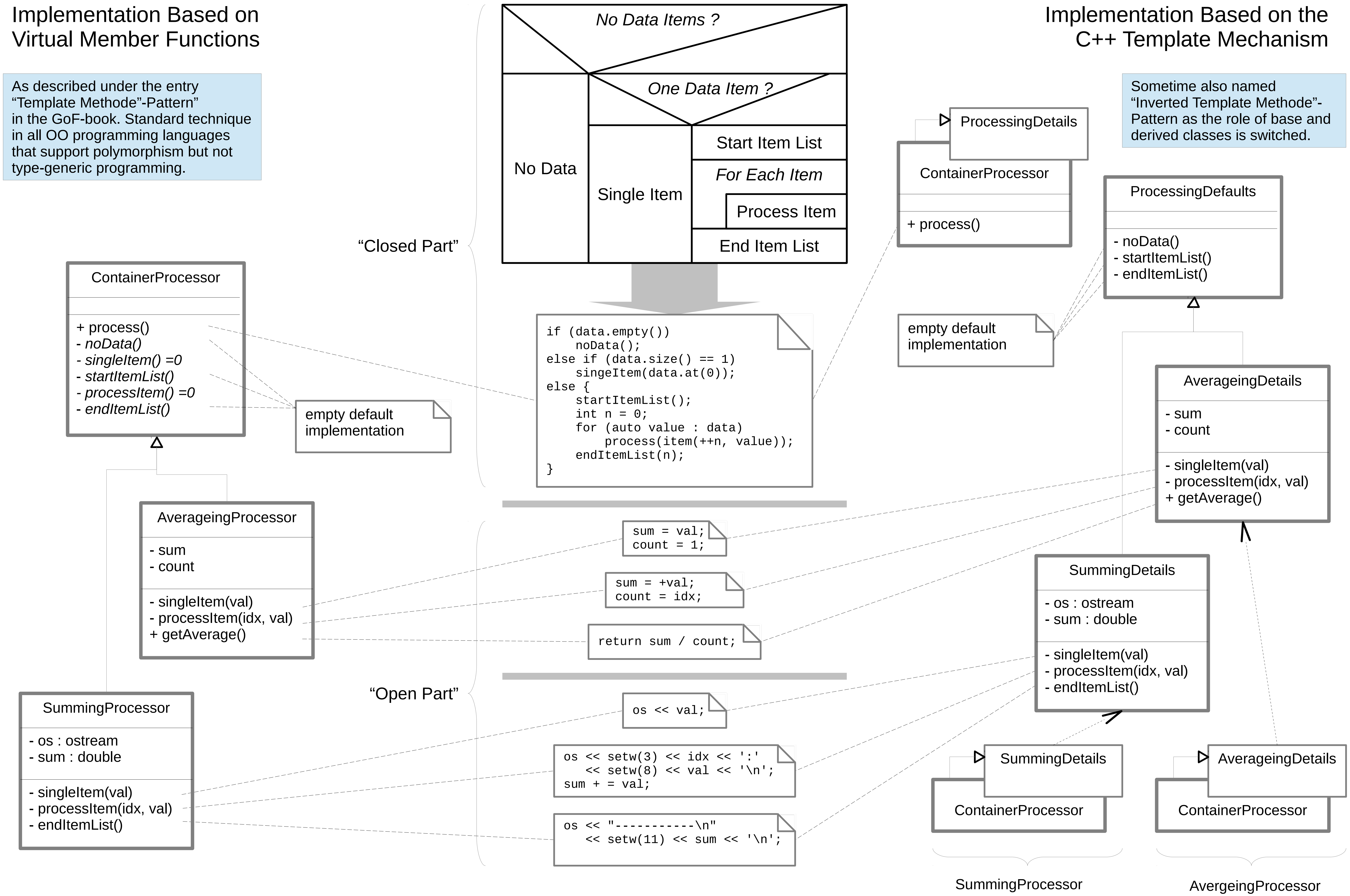
- Standard Design Patterns (from GoF) like
  - Composite
  - Template Methode
  - …
- Well-known C++ Idioms like
  - PIMPL (Pointer to Implementation)
  - RAII (Ressource Acquisition is Initialisation)
  - CRTP (Curiosly Recurring Template Pattern)
  - …
- Handy Little Techniques where useful
  - "Named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

But always judiciously decide … and Don't Overdo!

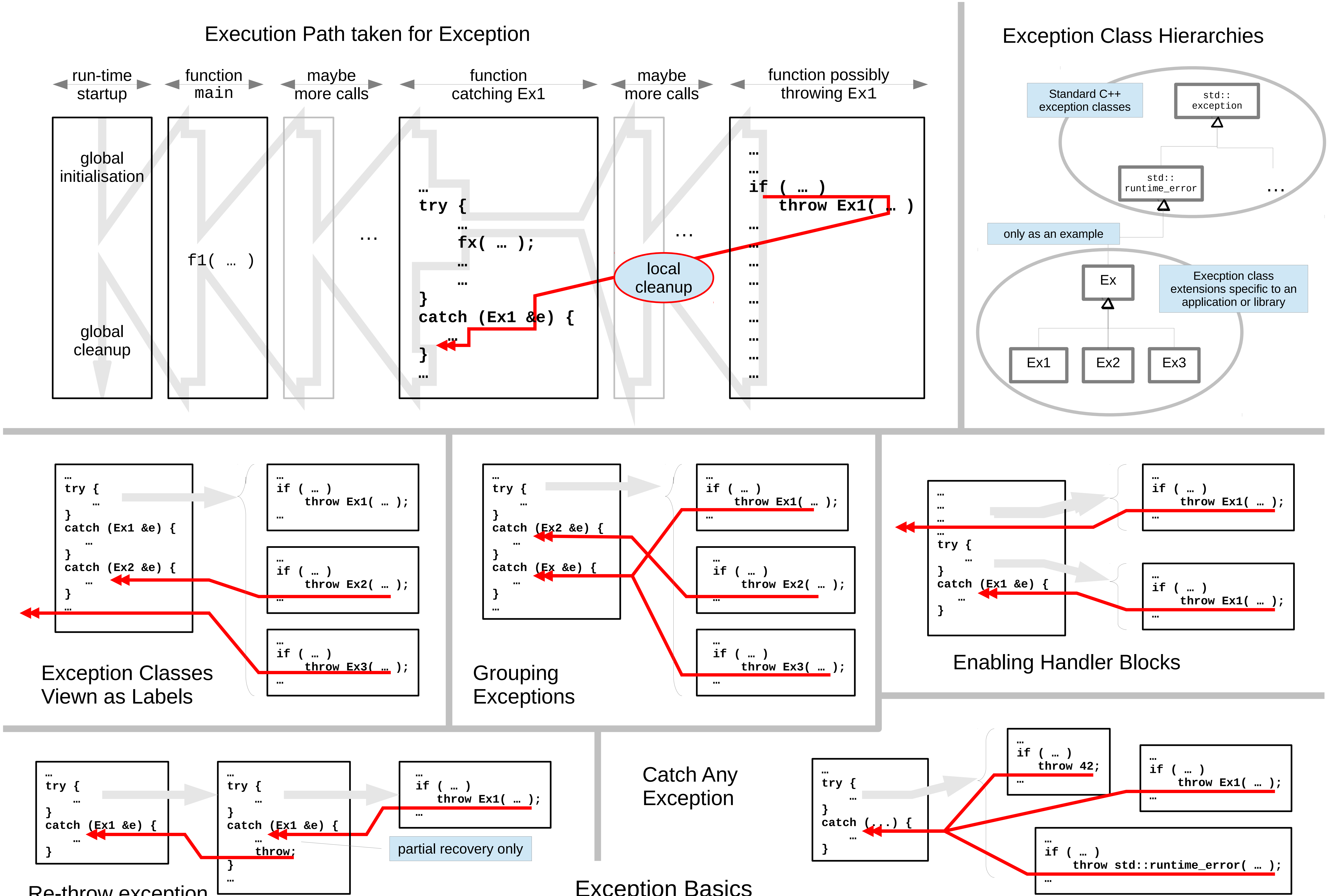
- **Not each and every global variable needs to be turned into a Singleton.**
- **Not each and every little config file needs to be parsed as full XML.**
- **Not each and every small class needs type genericity.**
- **...**

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!

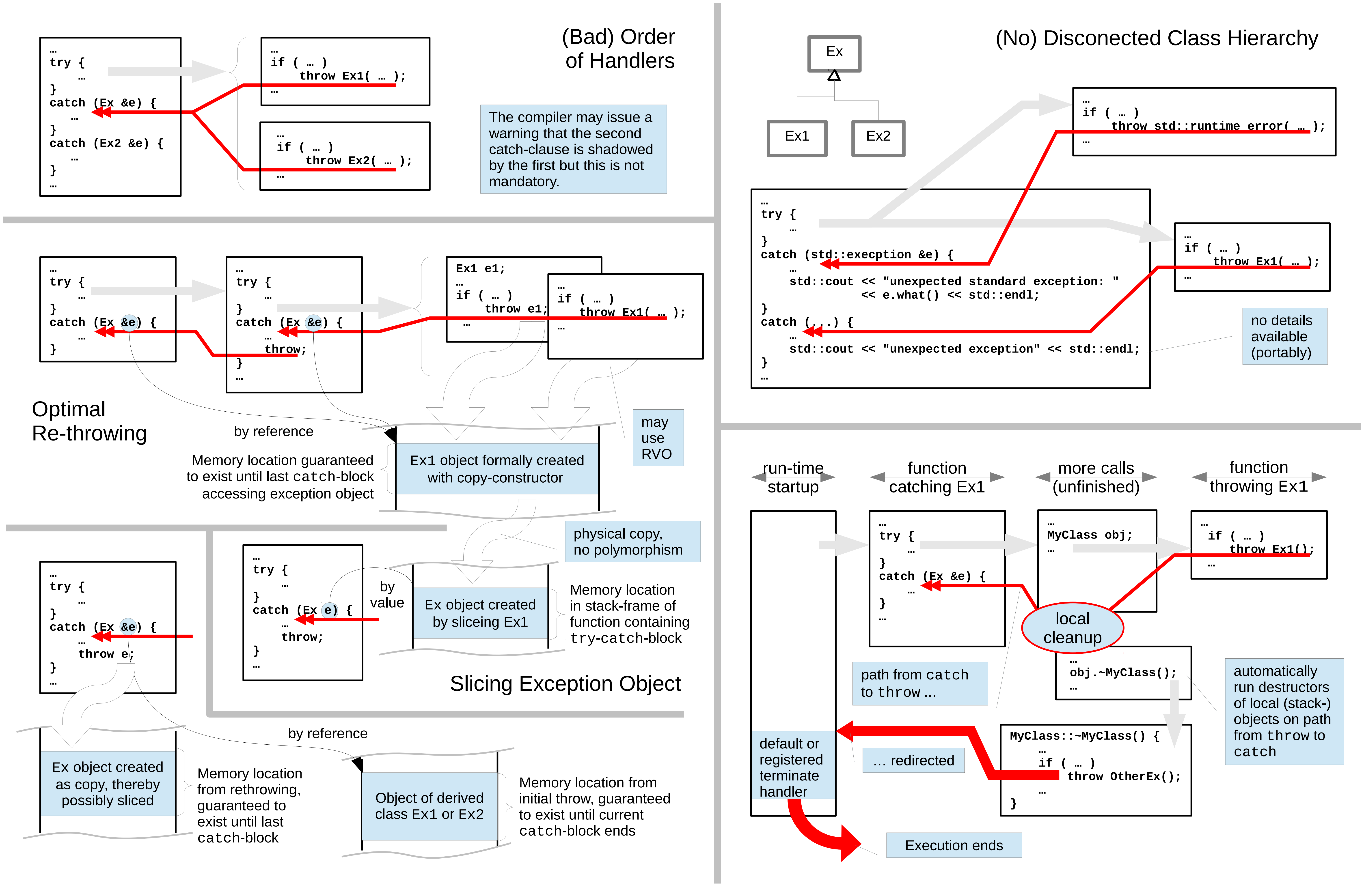
Guiding Principles



Example – “Open Close”-Principle



Exception Basics



Sub-optimal Re-throwing

(No) Throwing from Destructors

Exception Details

Classic Resource Management APIs

Principles	Examples: Unix/Linux – Processes	Examples: Unix/Linux – Files	Examples: C – Files	Examples: C *Free Memory (Heap)* / **C++** *Free Memory (Heap)*		Examples: **C++11** – `std::mutex m;`
Operation to acquire returns ...	`fork()`	`creat()`, `open()`	`fopen()`, `freopen()`	`malloc()`, `calloc()`, `realloc()` `new T`	 `new T[N]`	`m.lock()`, `m.try_lock()`
… 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 *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 adjacent memory locations like in a builtin array	no special return value (instead state of object is changed)
… 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 all builtin pointer operations		`m.native_handle()`
… until final release (eventually returning resource to a pool)	`wait()`, `waitpid()`	`close()`	`fclose()`	`free()` `delete …`	 `delete[] …`	`m.unlock()`
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);
}
```

Acquire Resource during Execution of Code Segment

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

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

Acquire Resource for Lifetime of Object

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

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

Wrapped Resource

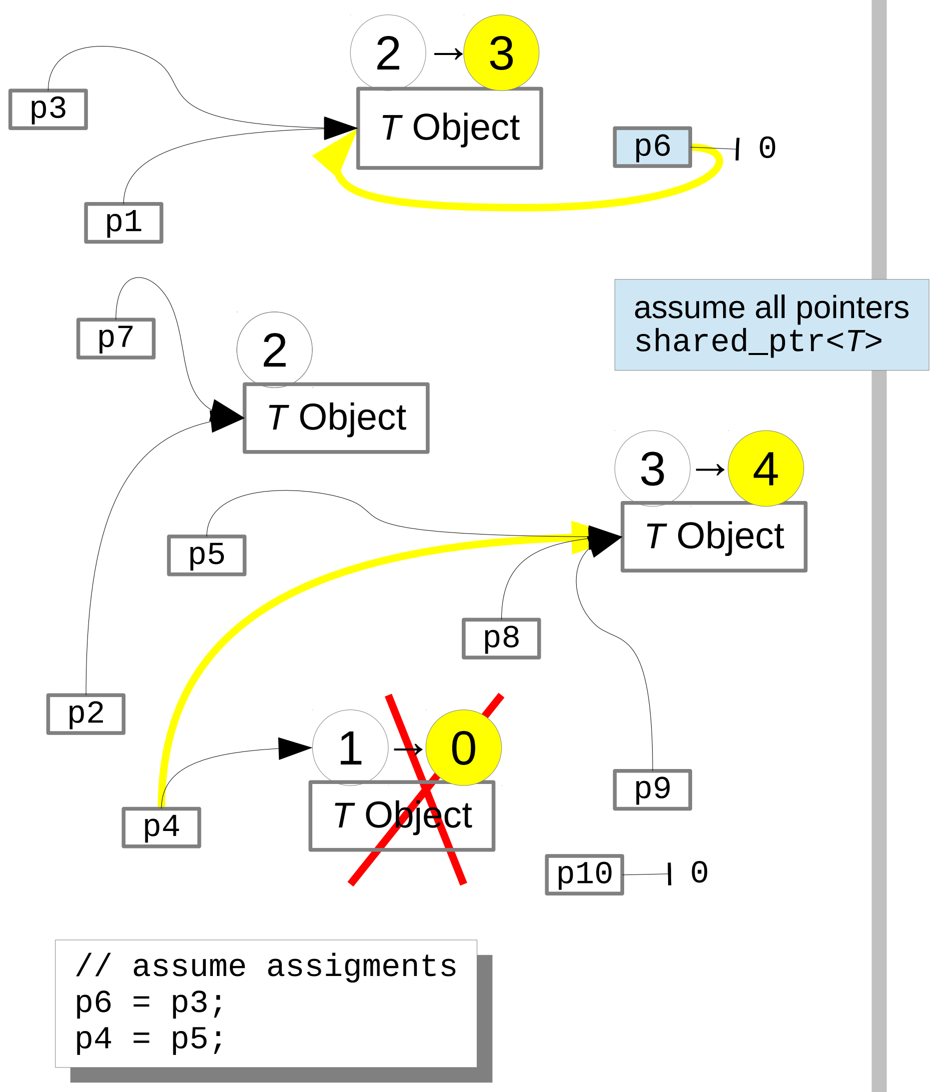
Optionally add Convenience Operations

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

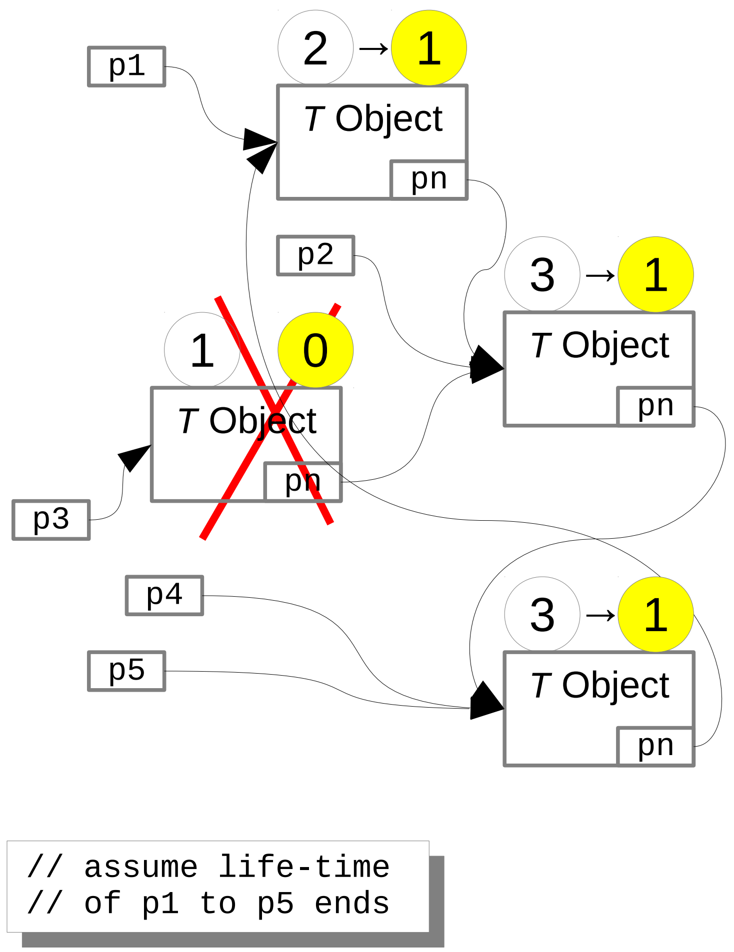
Easy and Secure Use via Automatic Conversion

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

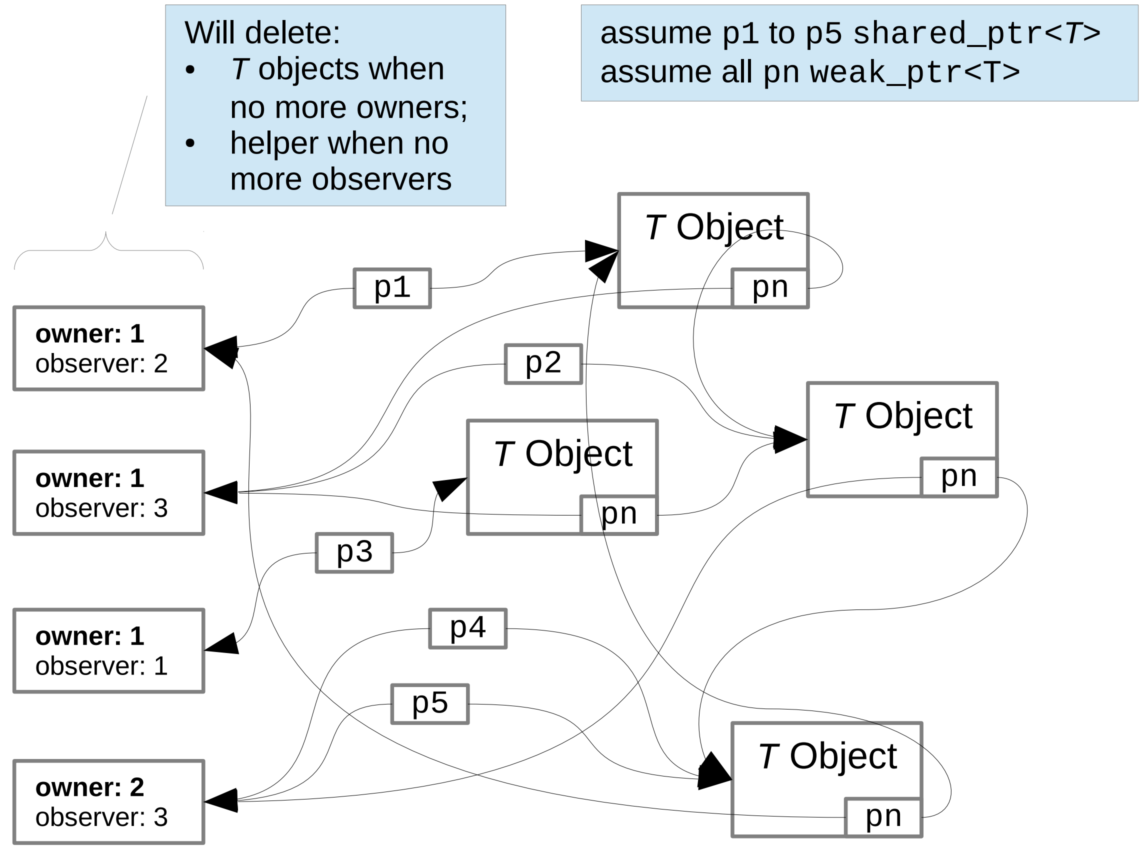
Classic Resource Management vs. RAII



Reference Counting Principle

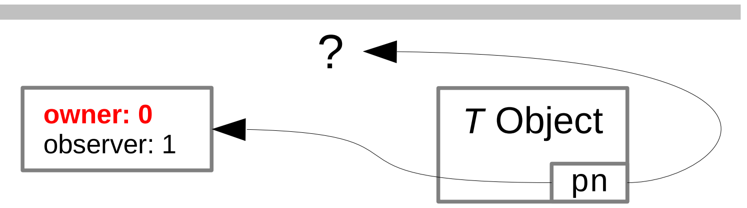


Problem of Cyclic References



Breaking Cycles by using Weak Pointers

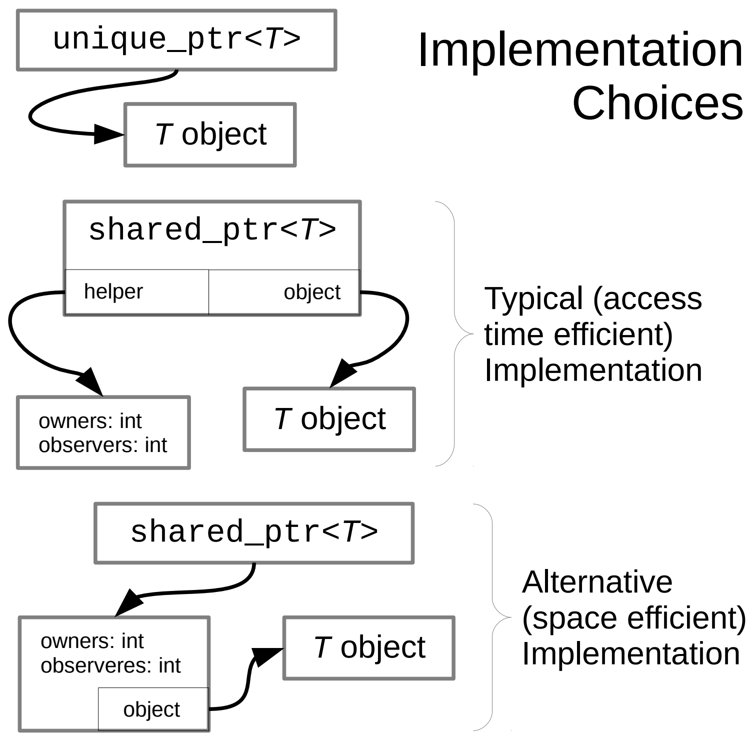
Dangling Weak Pointer



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 Assignment	yes		
Copy Assignment	no*		a *T* 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	

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

Implementation Choices



Smart Pointer Comparison