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;
```

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 {
        …
}
```

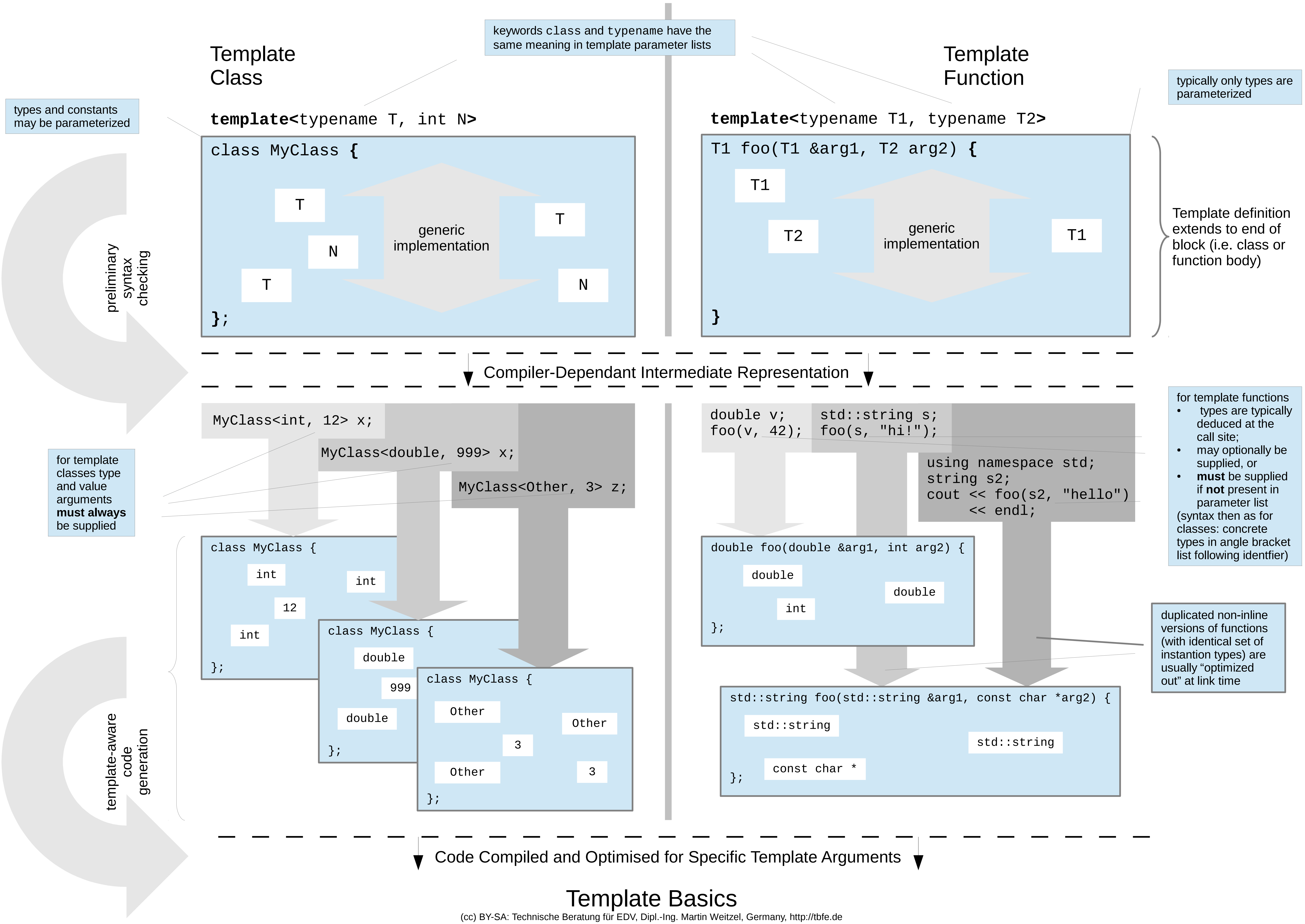
```
RingBuffer<100> b;
RingBuffer<30> b2;
```

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.

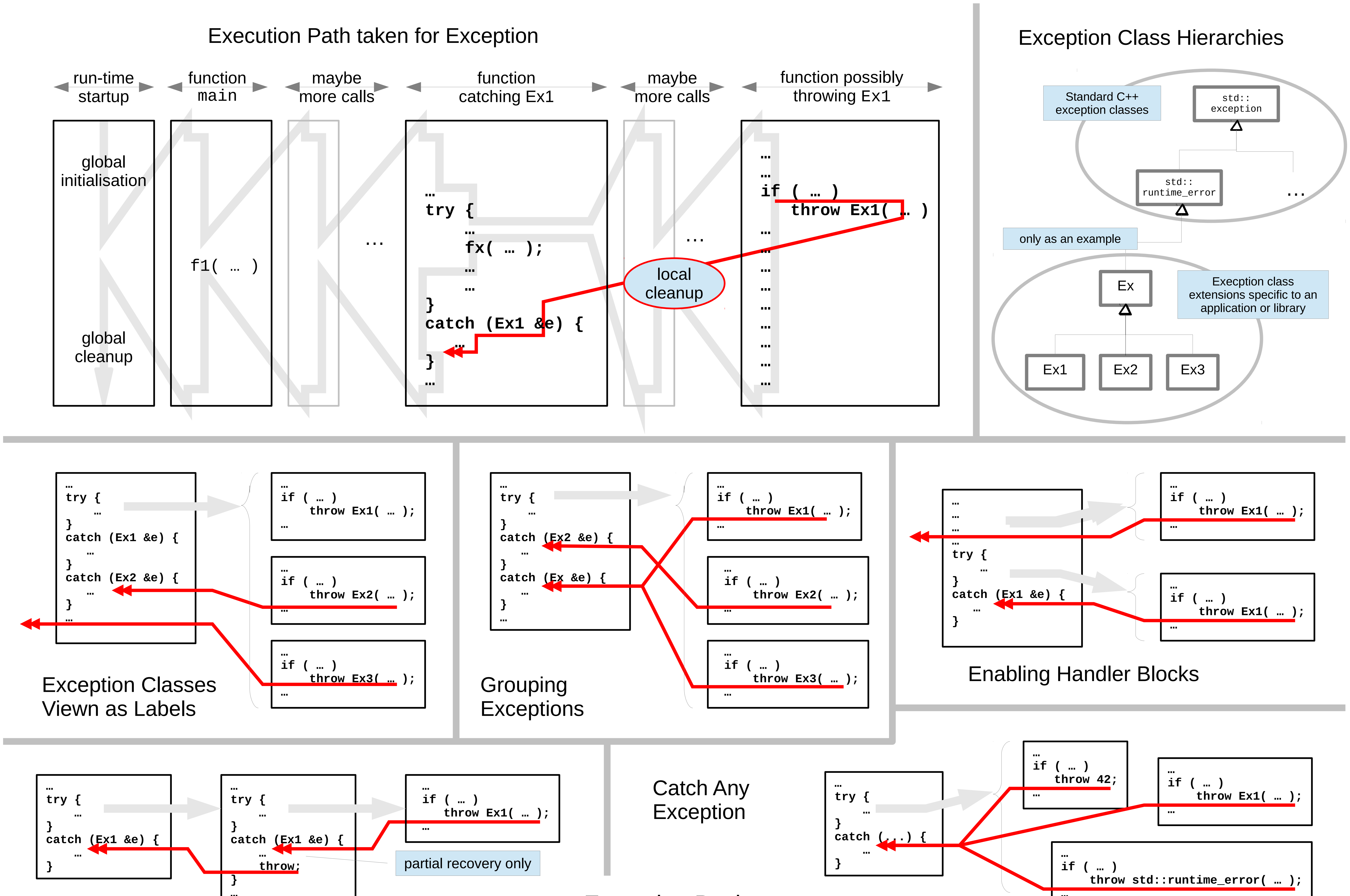
```
template<typename T, std::size_t N>
class RingBuffer {
     T data[N+1];
protected:
     std::size_t iput;
     std::size_t iget;
     static std::size_t wrap(std::size_t idx) {
          return idx % (N+1);
     }
public:
     RingBuffer()
          : iput(0), iget(0)
     {}
     bool empty() const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

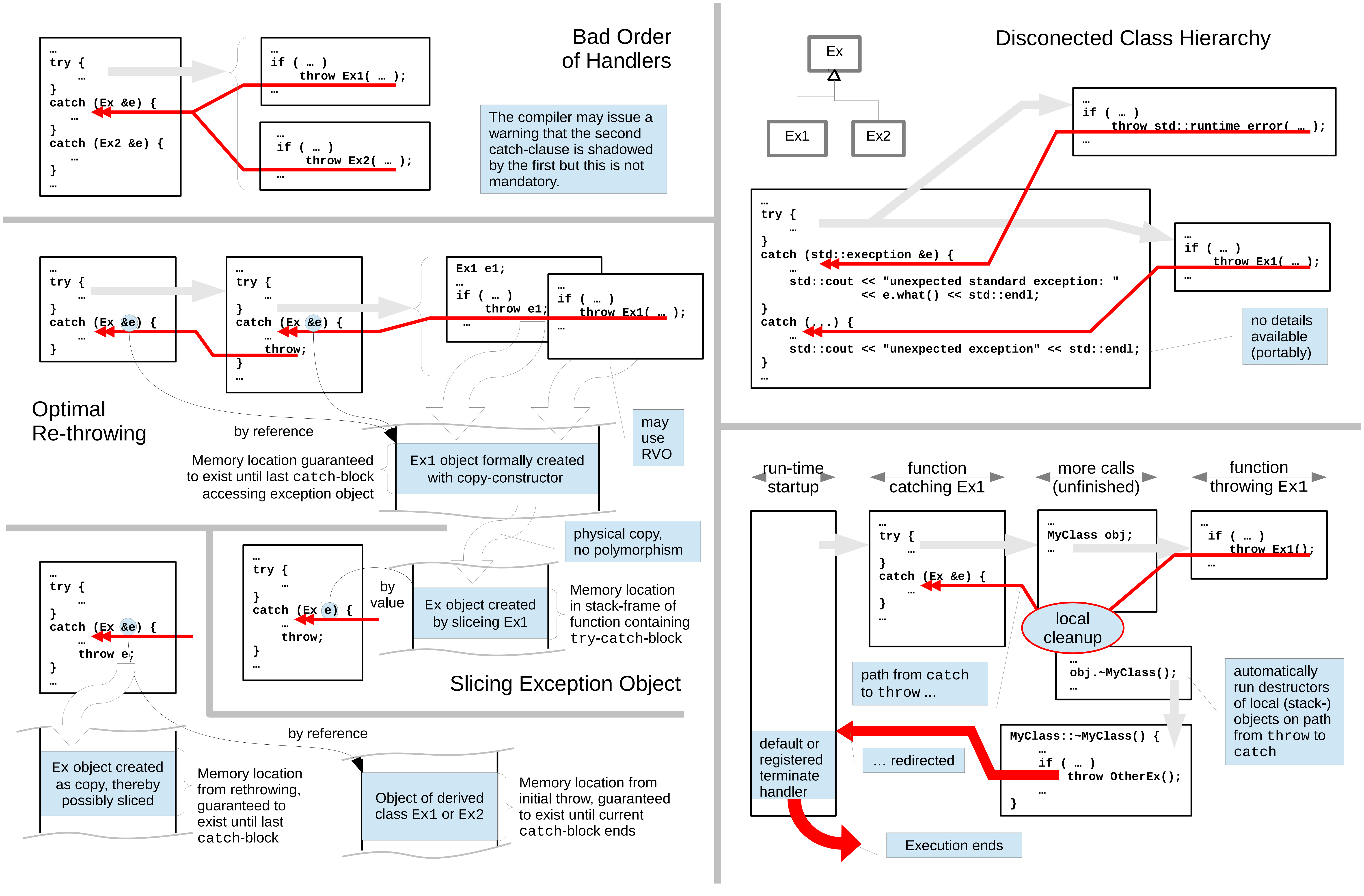
Parametrizing Types and Sizes



Template Basics



Exception Basics



Sub-optimal Re-throwing

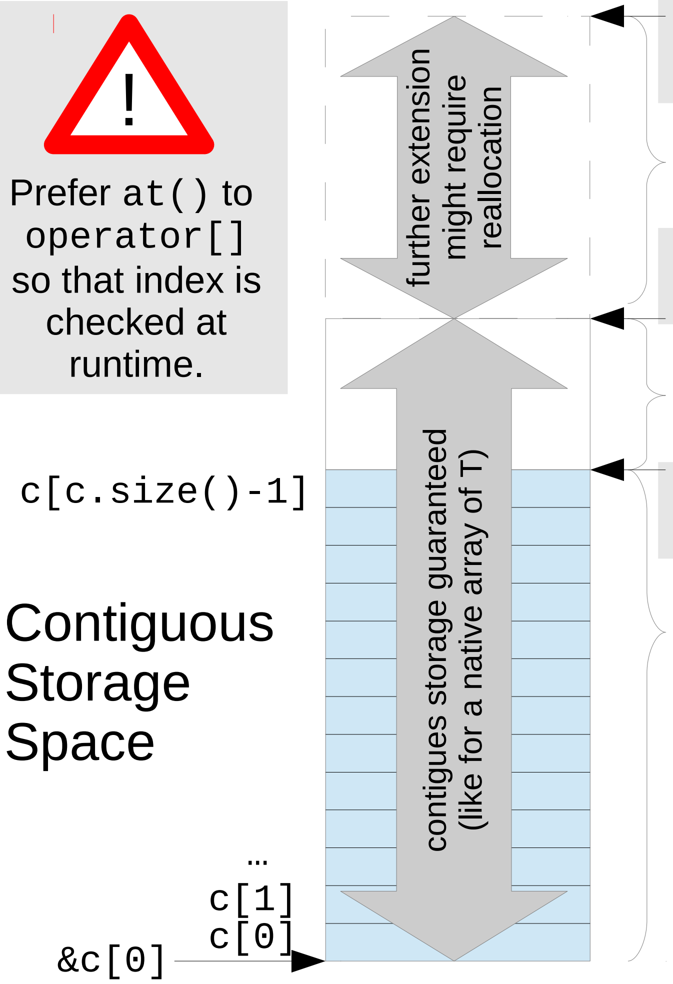
Exception Details

Throwing from Destructors

std::vector<T> c;

Contiguous Storage Space

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()`

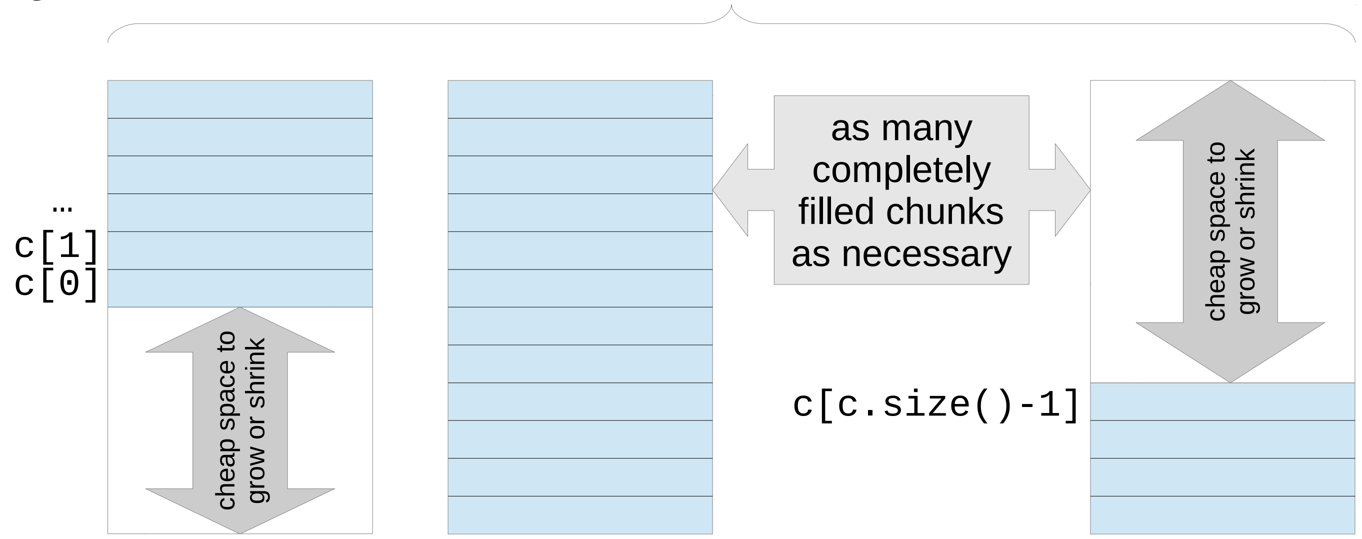
Direct element access (in addition to dereference):
- **pointer arithmetic only**

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

std::deque<T> c;

Double-Ended Queue

`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

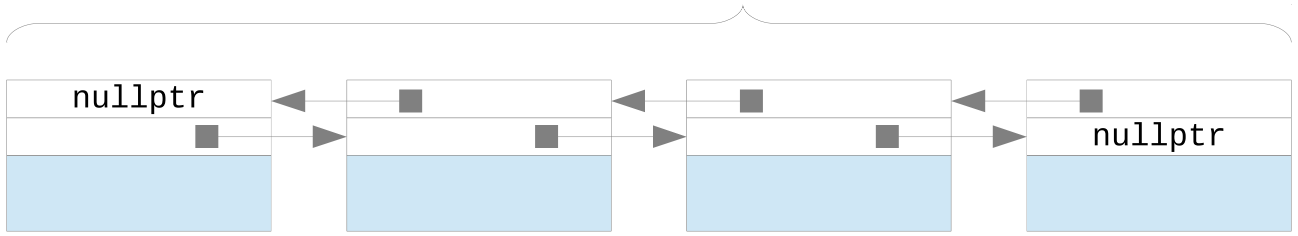
Direct element access (in addition to dereference):
- presumably some “masking and shifting”
- indirect memory access
- address arithmetic

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

std::list<T> c;

Double Linked List

`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

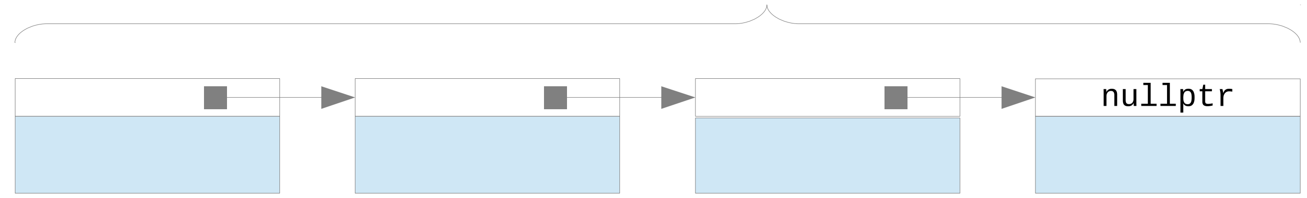
Direct element access not supported!
Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by assignment

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

std::forward_list<T> c;

Singly Linked List

objects of type `T` initialised with constructor



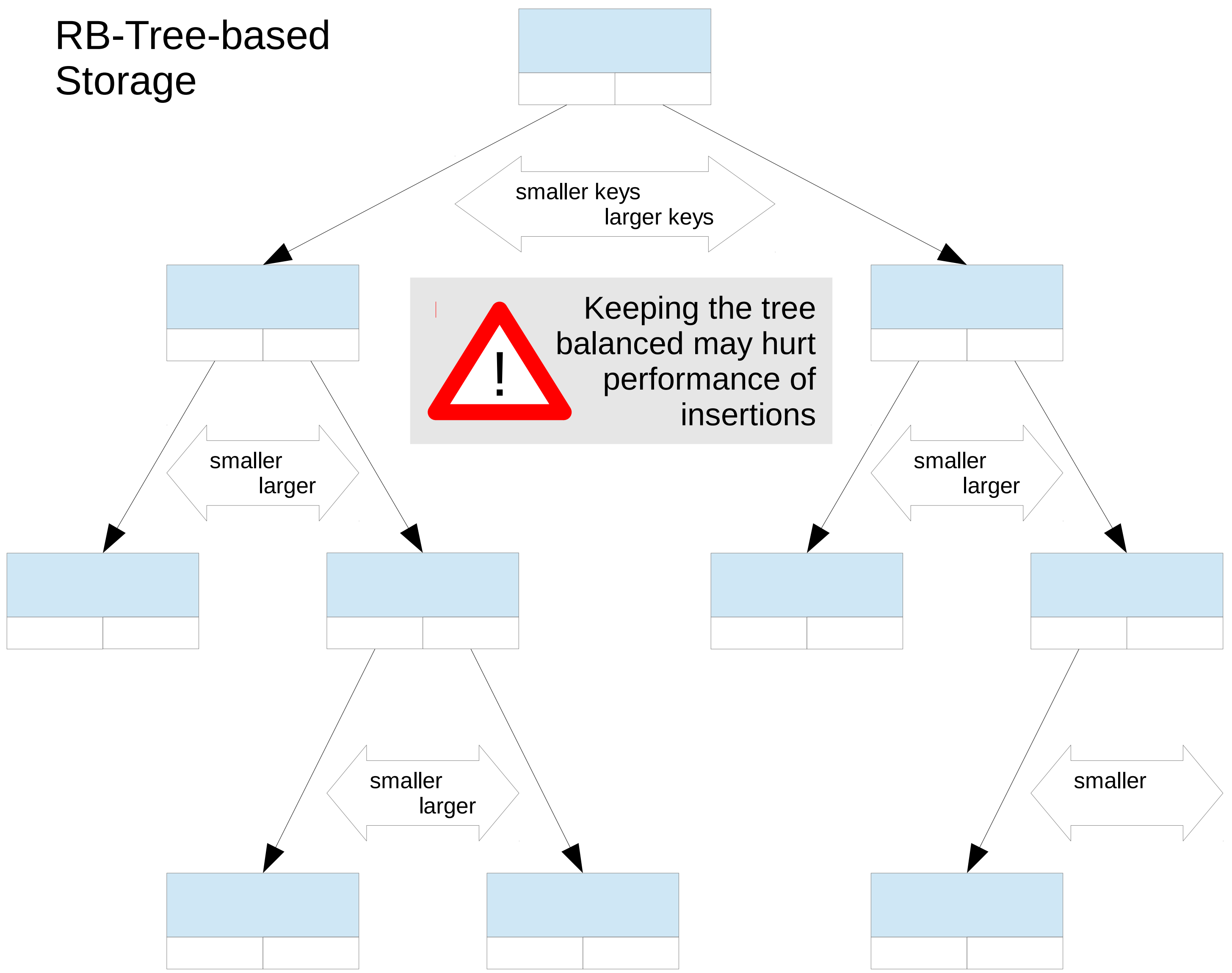
Use `c.empty()` to check wether elements exist.

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

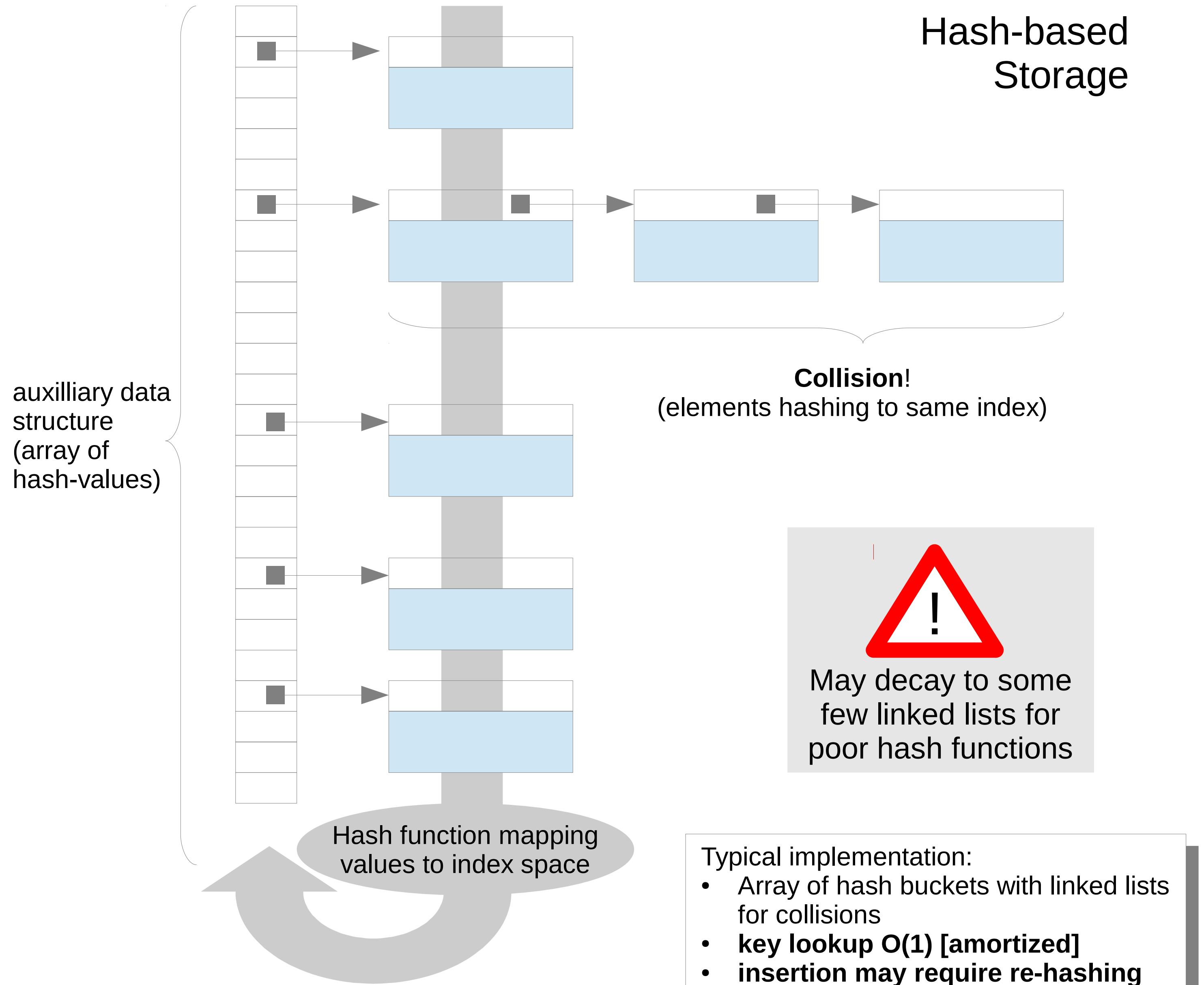
Direct element access not supported!
Minimal implementation of iterator: one pointer
Advancing iterators requires memory access followed by assignment

STL – Sequence Container Classes

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



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

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.

STL – Associative Container Classes

Container Dimension

Library	STL									Standard Strings	Iterator Interface to I/O-Streams		e.g. Boost	Others
Kind of Container	Sequential Containers					Associative Containers							Special Containers • `ptr_vector` • `ptr_set` • … More Maps • `bimap` • `multi_index` • …	
Data Structure	*Random Access*			*Sequential Access*		*Tree*	*Hash*	*Tree*	*Hash*		*I/O operations for some type T*			
Class Name	`array`	`vector`	`deque`	`list`	`forward_list`	`set` `multiset`	`unordered_set` `unordered_multiset`	`map` `multimap`	`unordered_map` `unordered_multimap`	`string` `wstring` …	`istream_iterator`	`ostream_iterator`		
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

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()`

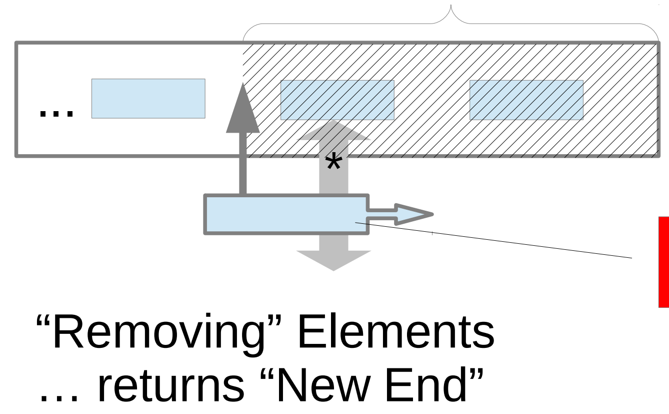
Algorithm Dimension		
	STL	Access: • `find` • `search` • …
		Modify: • `remove` • `sort` • …
		Miscellaneous: • `count` • `mimmax_seq` • …
	e.g. Boost	Algorithm: • `join` • … String_algo: • `trimleft` • `trimright` • … …
	Others	

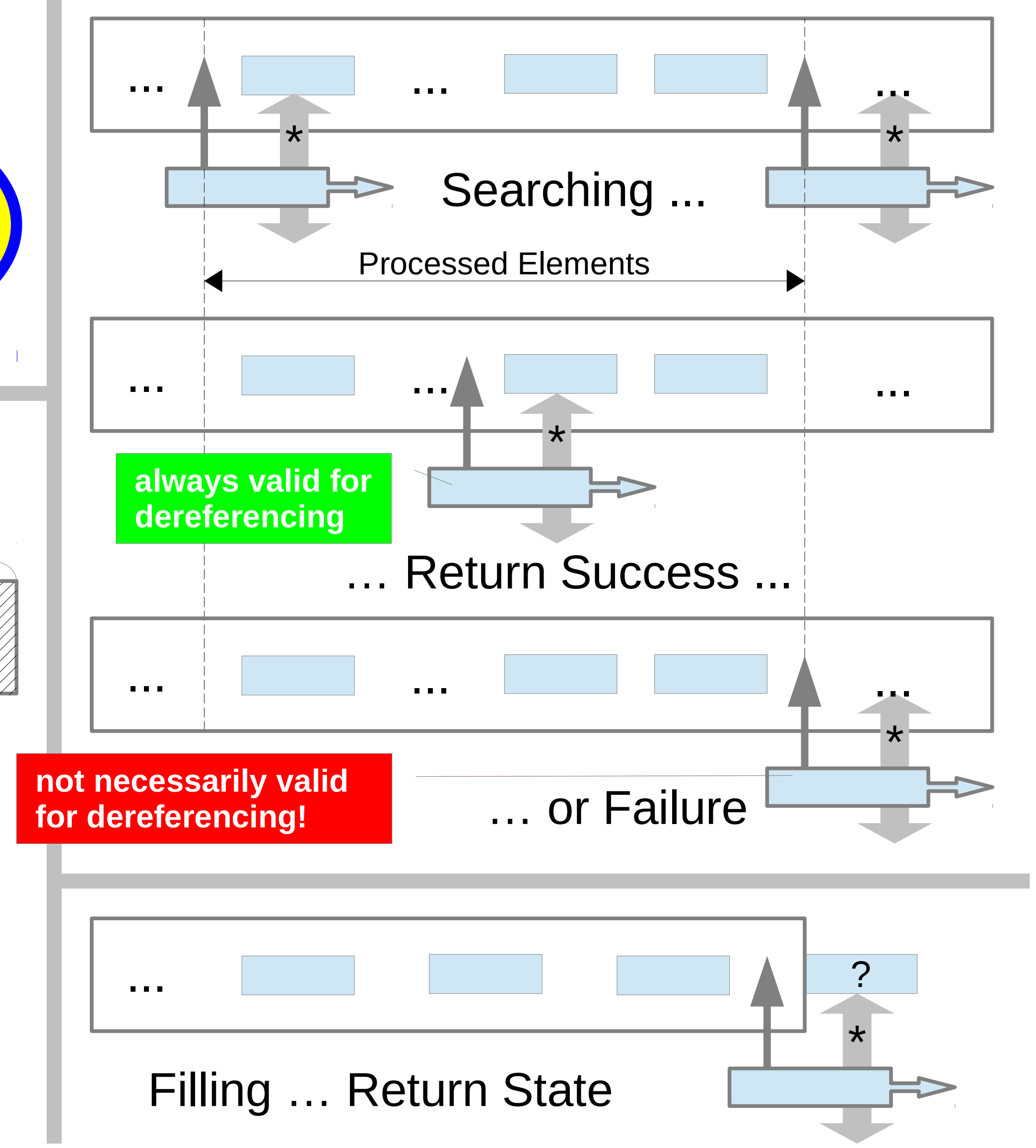
operations expected from iterators

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.





Input Iterators Semantic Restrictions

*	must only be used for <u>read</u> access
++	must follow <u>each</u> read exactly once

Output Iterators Semantic Restrictions

*	must only be used for <u>write</u> access
++	must follow <u>each</u> write exactly once

STL – Iterator Usages

Code-Units vs. Code-Points

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

- 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.

Visual Appearance (as perceived by user)

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/)

] ¤ € Ü e

no precomposed form

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

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

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 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.

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!

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

convert to every builtin **integral** type

convert to every 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 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`, …

C++11

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

Advanced String Operations

Advanced operations:

- too many to list (→ RTFM).

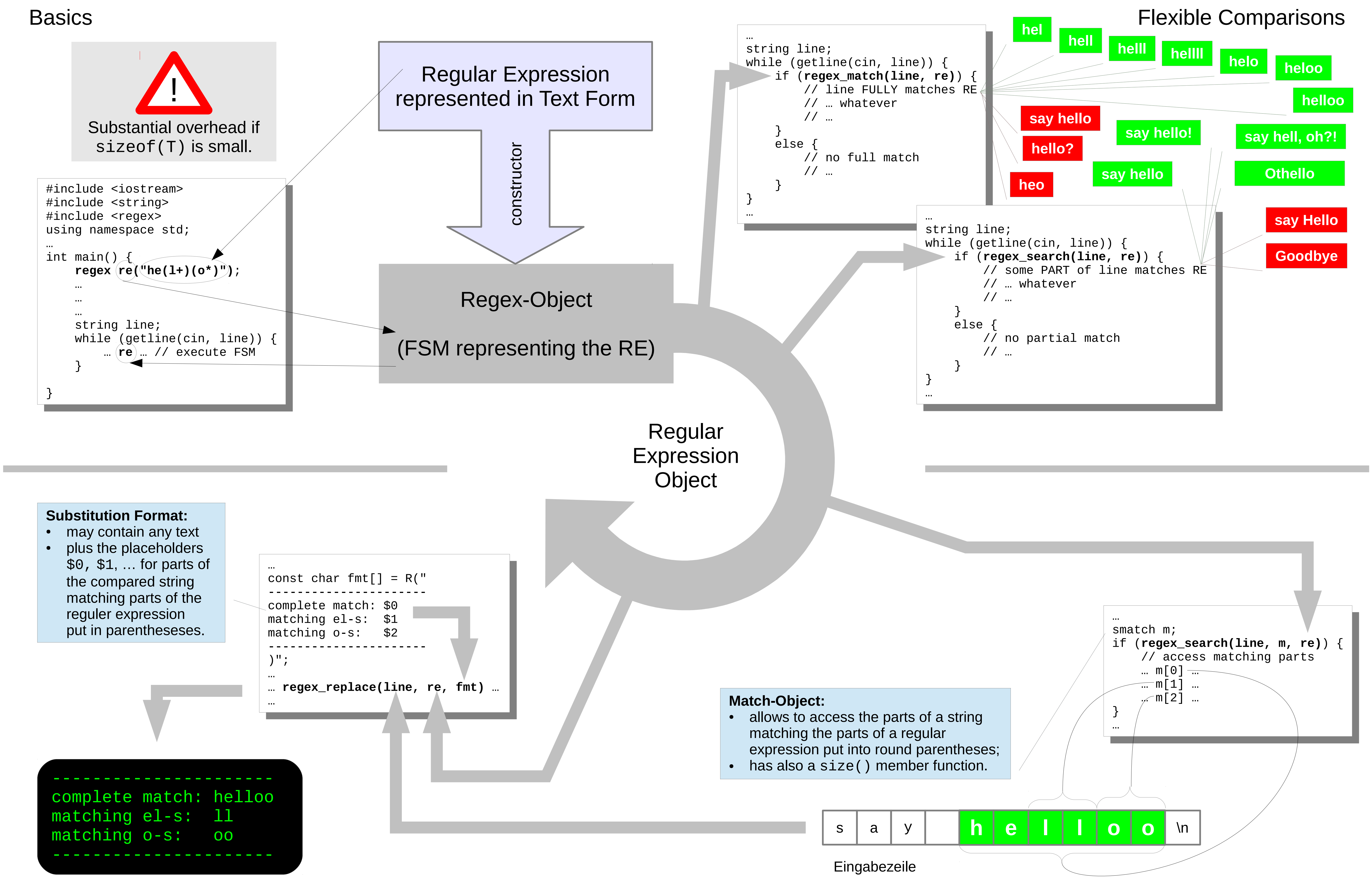
Process like an STL container:

- 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`

Standard Strings 101



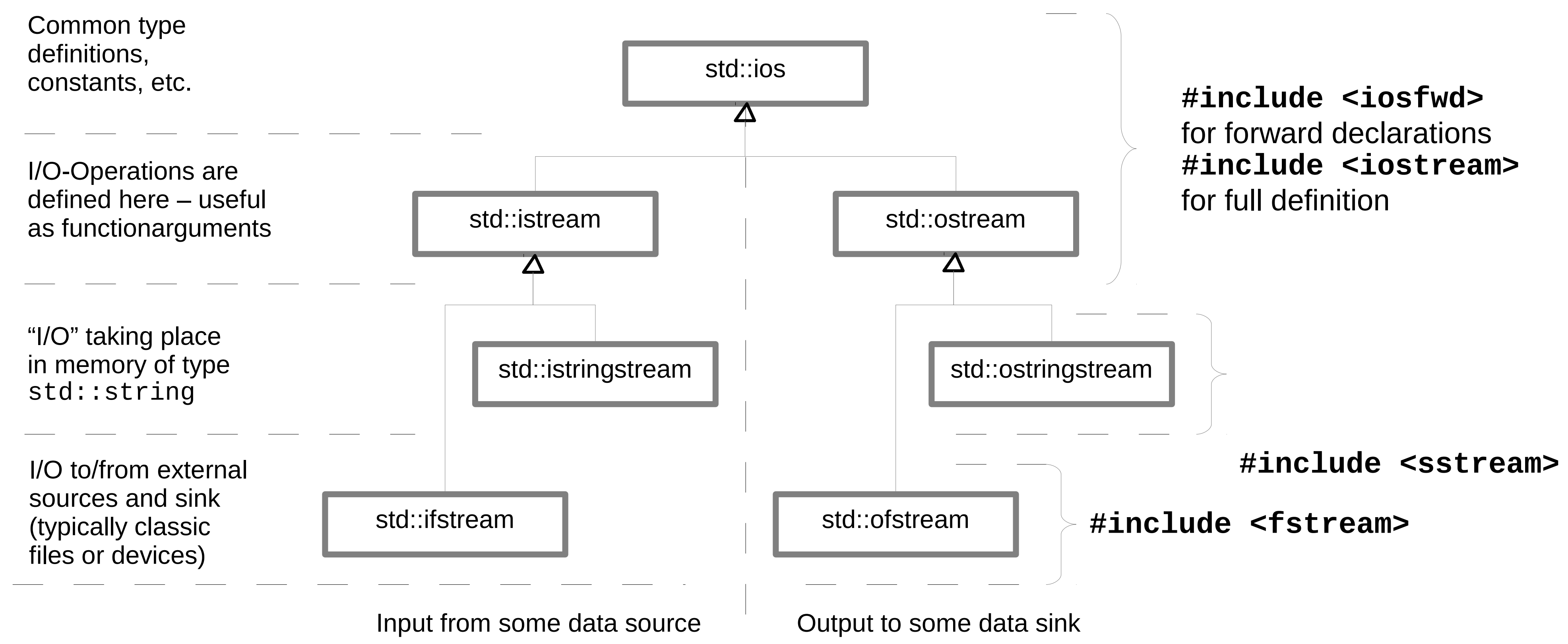
Powerful Substitutions

Regular Expressions

Easy Parsings

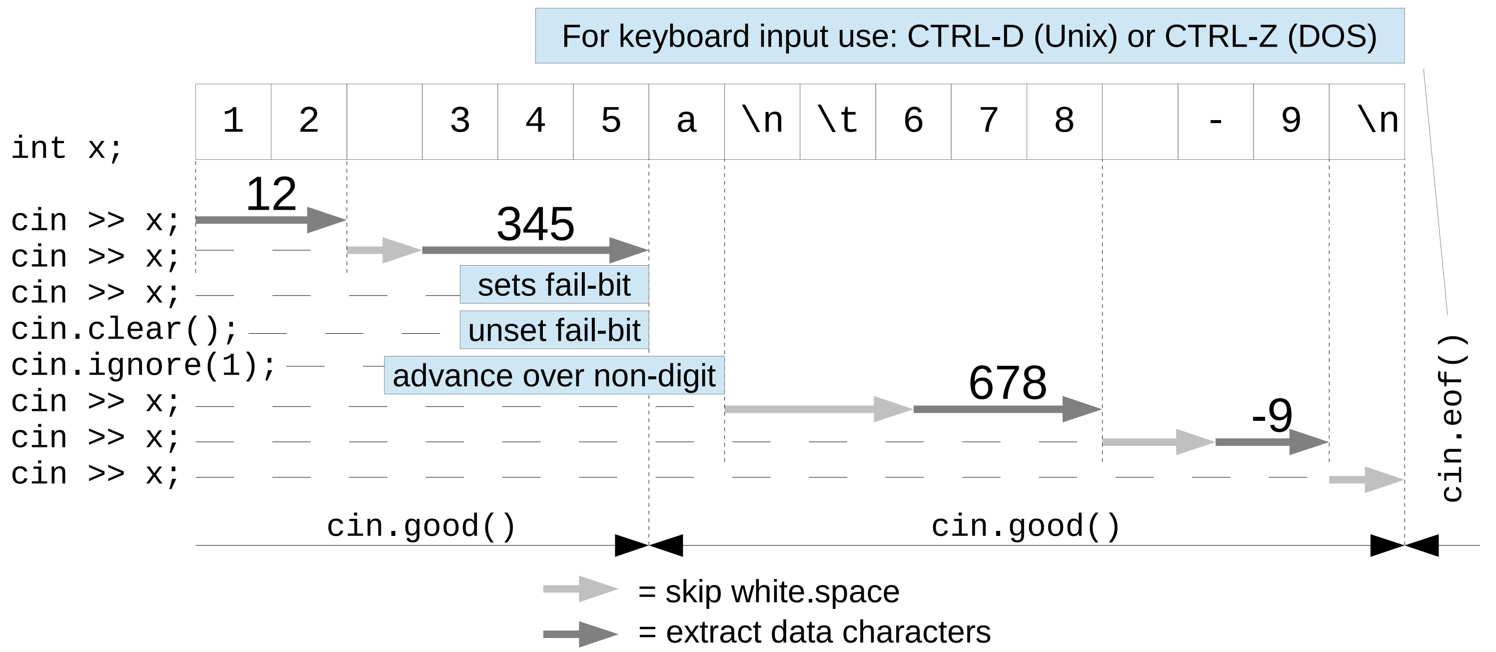
I/O-Stream Basics

I/O-Streams Front-End

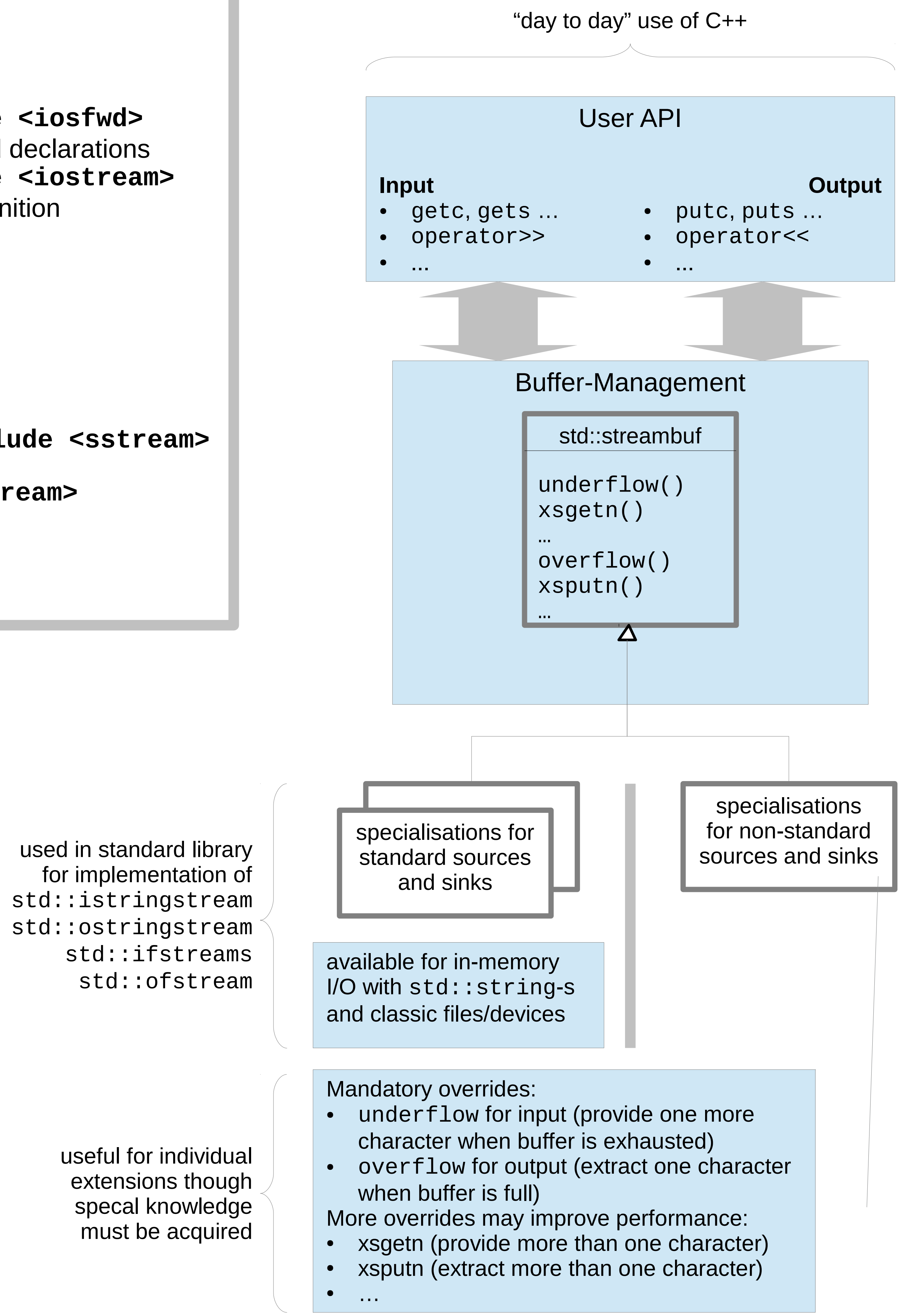


I/O-Stream States (assuming namespace std and stream named *s*)

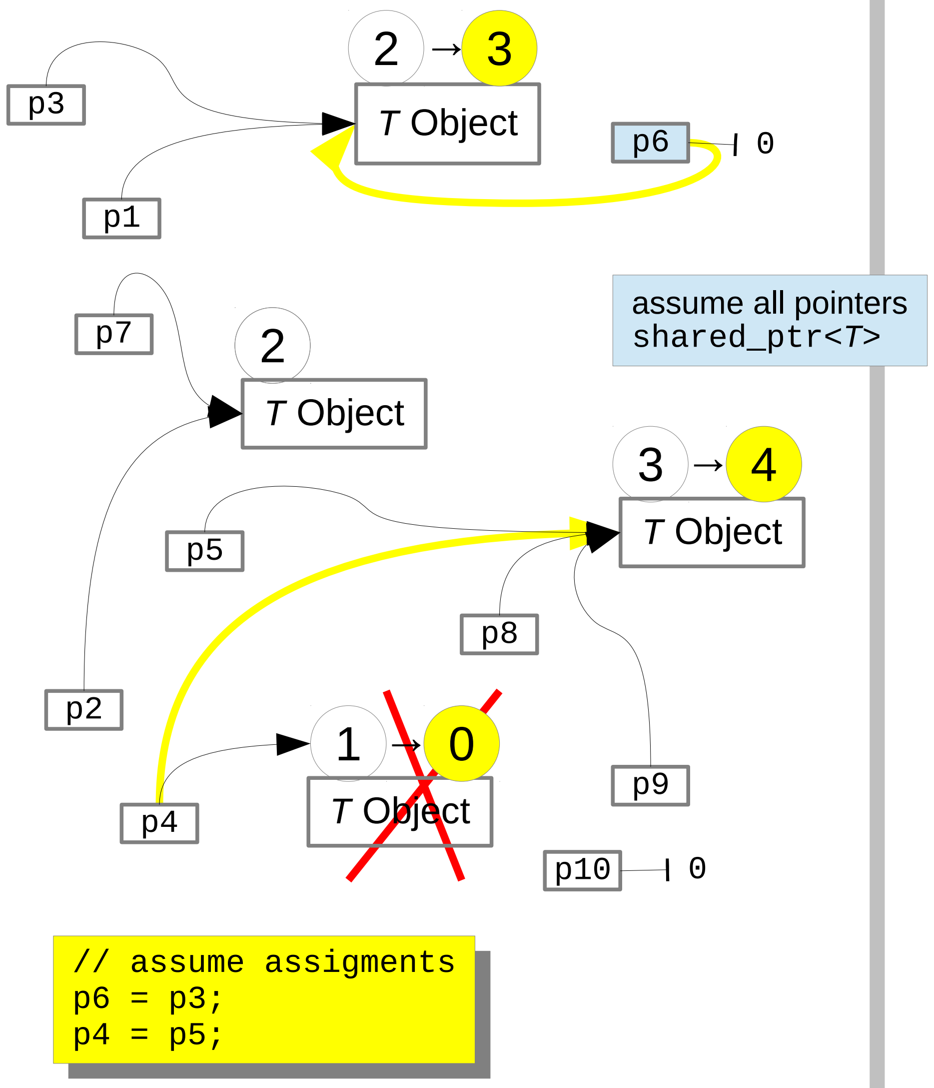
Set ...	Name	is set ?	set explicitely	all unset ?	unset all
… on format error	`ios::failbit`	*s*`.fail()`	*s*`.clear(ios::failbit)`		
… on end of input	`ios::eofbit`	*s*`.eof()`	*s*`.clear(ios::eofbit)`	*s*`.good()`	*s*`.clear()`
(implem. defined)	`ios::badbit`	*s*`.bad()`	*s*`.clear(ios::badbit)`		



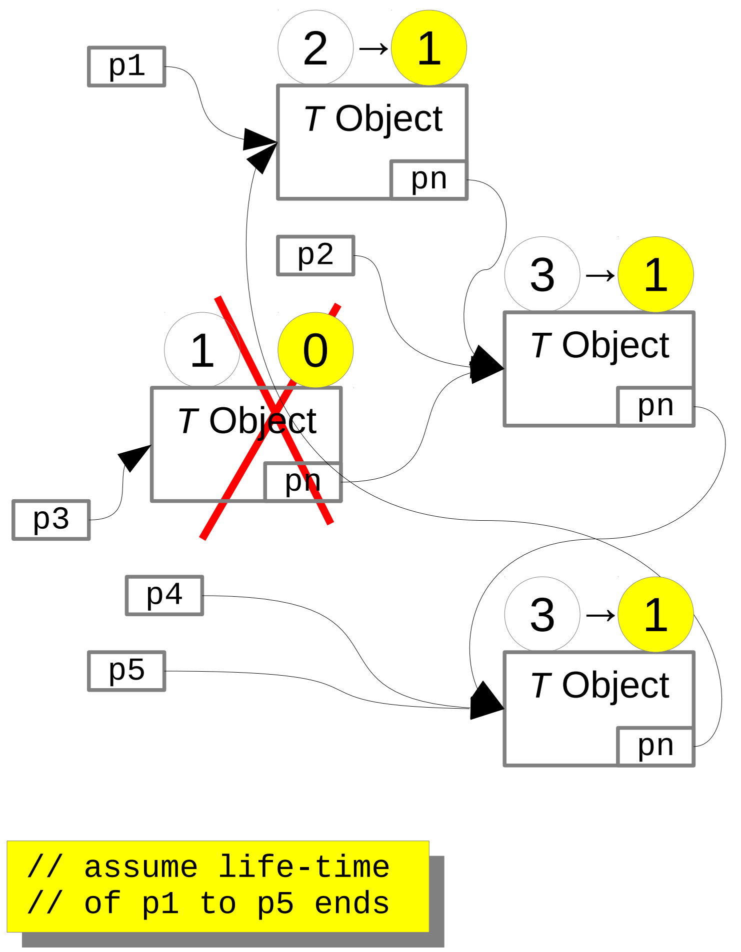
I/O-Streams State-Bits



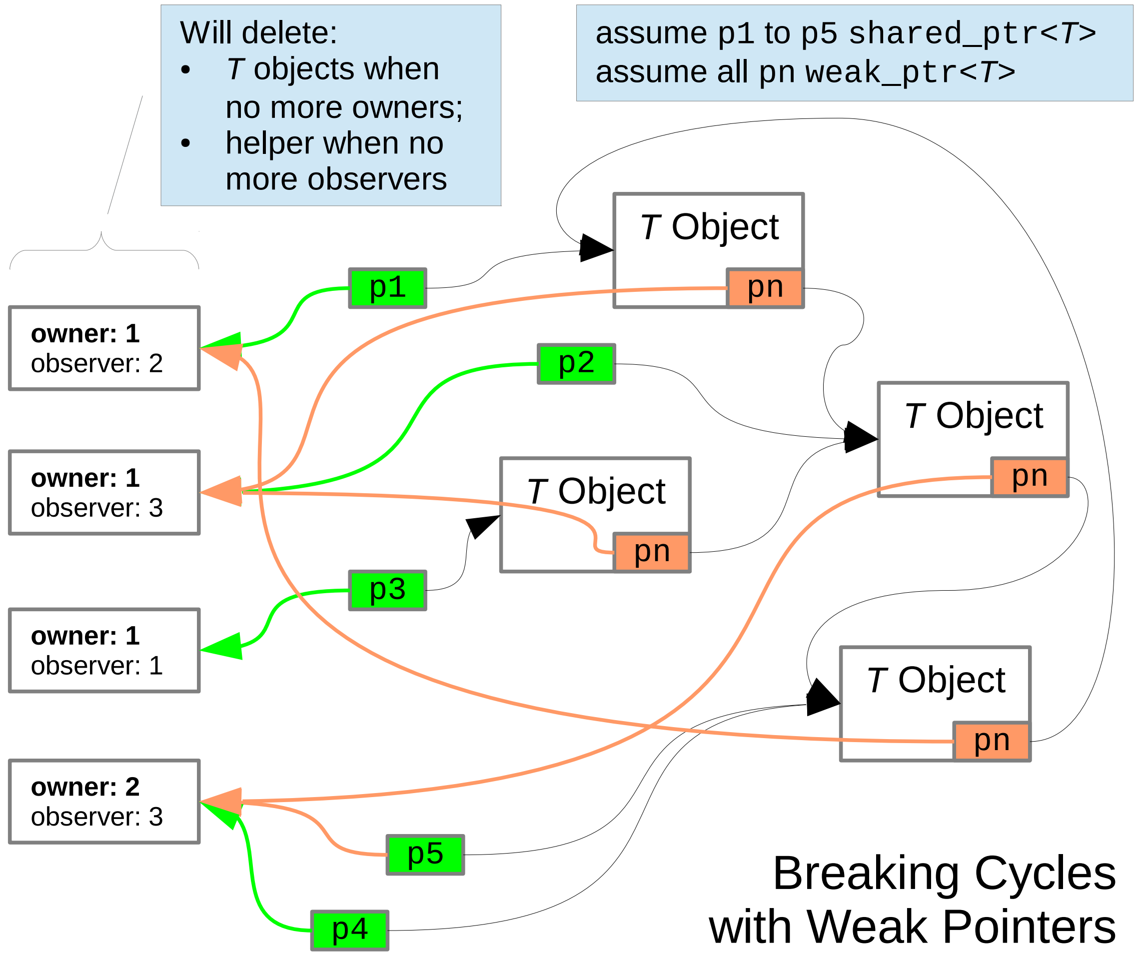
I/O-Streams Back-End



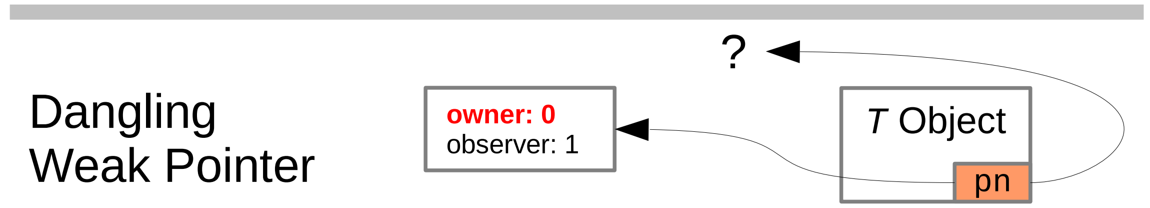
Reference Counting Principle



Problem of Cyclic References



Breaking Cycles with Weak Pointers

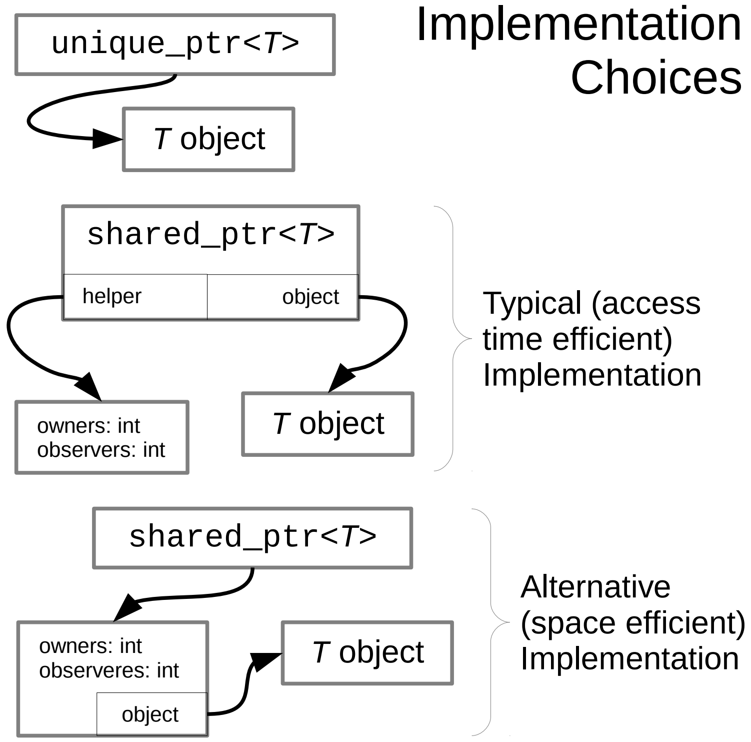


Dangling Weak Pointer

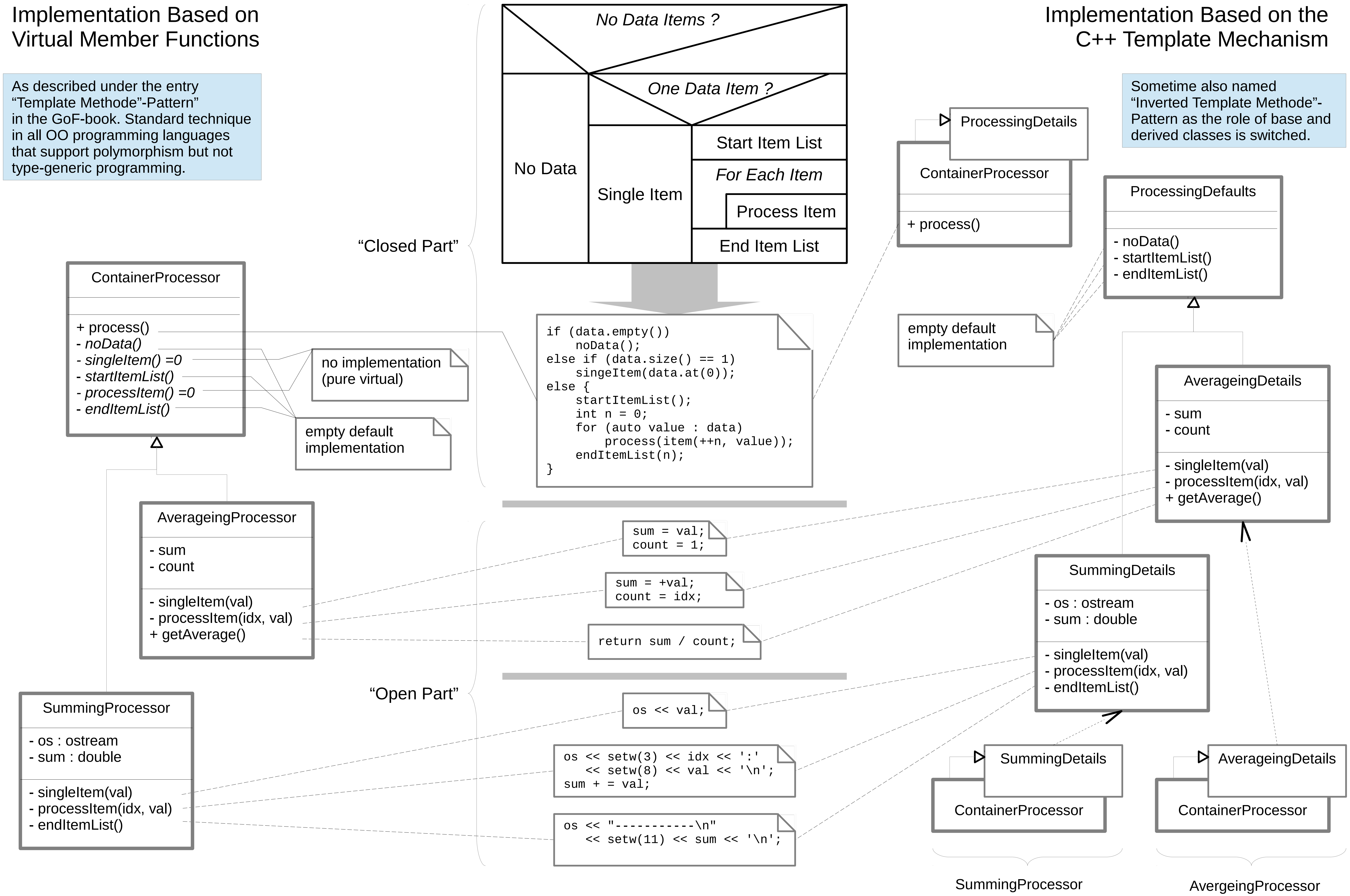
Comparing ...	**std::unique_ptr<*T*>**	**std::shared_ptr<*T*>**	**Remarks**
Characteristic	refers to a single object of type *T*, **uniquely owned**	refers to a single object of type *T*, **possibly shared with other referrers**	may also refer to "no object" (like a `nullptr`)
Data Size	same as plain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

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

Implementation Choices



Smart Pointer Comparison



Example – “Open Close”-Principle