

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

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

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

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

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

Parametrizing *Type*

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

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

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

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

RingBuffer<double> b;
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.

```
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;

RingBuffer b;

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

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

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

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

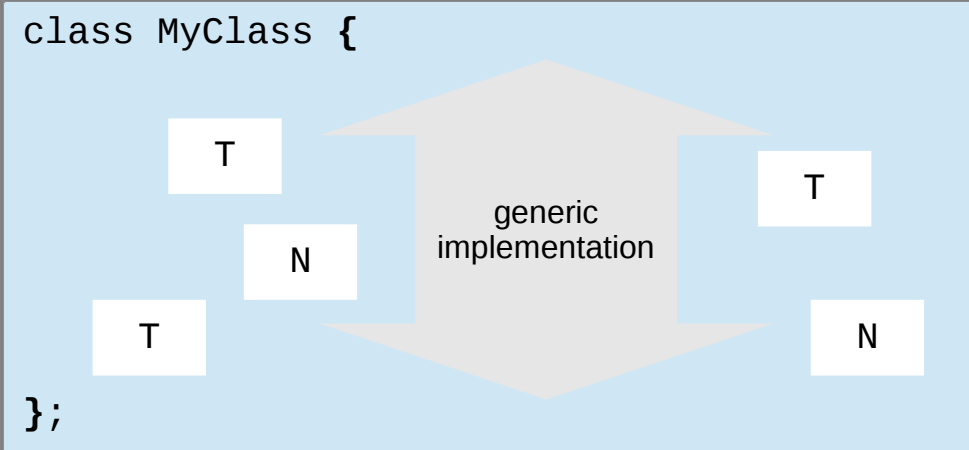
RingBuffer<double, 10> b;
RingBuffer<int, 10000> x;
RingBuffer<string, 42> y;
RingBuffer<MyClass, 9> z;

Parametrizing Types and Sizes

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

Template Class

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

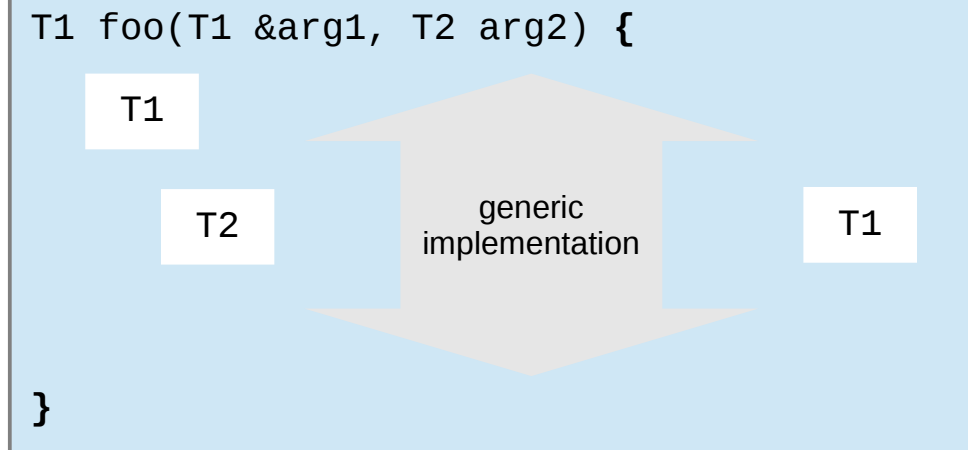


types and constants may be parameterized

preliminary
syntax
checking

Template Function

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



typically only types are parameterized

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

Compiler-Dependant Intermediate Representation

```
MyClass<int, 12> x;
```

```
MyClass<double, 999> x;
```

```
MyClass<Other, 3> z;
```

```
class MyClass {  
    int  
    int  
    12  
    int  
};
```

```
class MyClass {  
    double  
    999  
    double  
};
```

```
class MyClass {  
    Other  
    3  
    Other  
    3  
};
```

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

template-aware
code
generation

```
double v; std::string s;  
foo(v, 42); foo(s, "hi!");
```

```
using namespace std;  
string s2;  
cout << foo(s2, "hello")  
      << endl;
```

```
double foo(double &arg1, int arg2) {  
    double  
    int  
    double  
};
```

```
std::string foo(std::string &arg1, const char *arg2) {  
    std::string  
    const char *  
    std::string  
};
```

for template functions

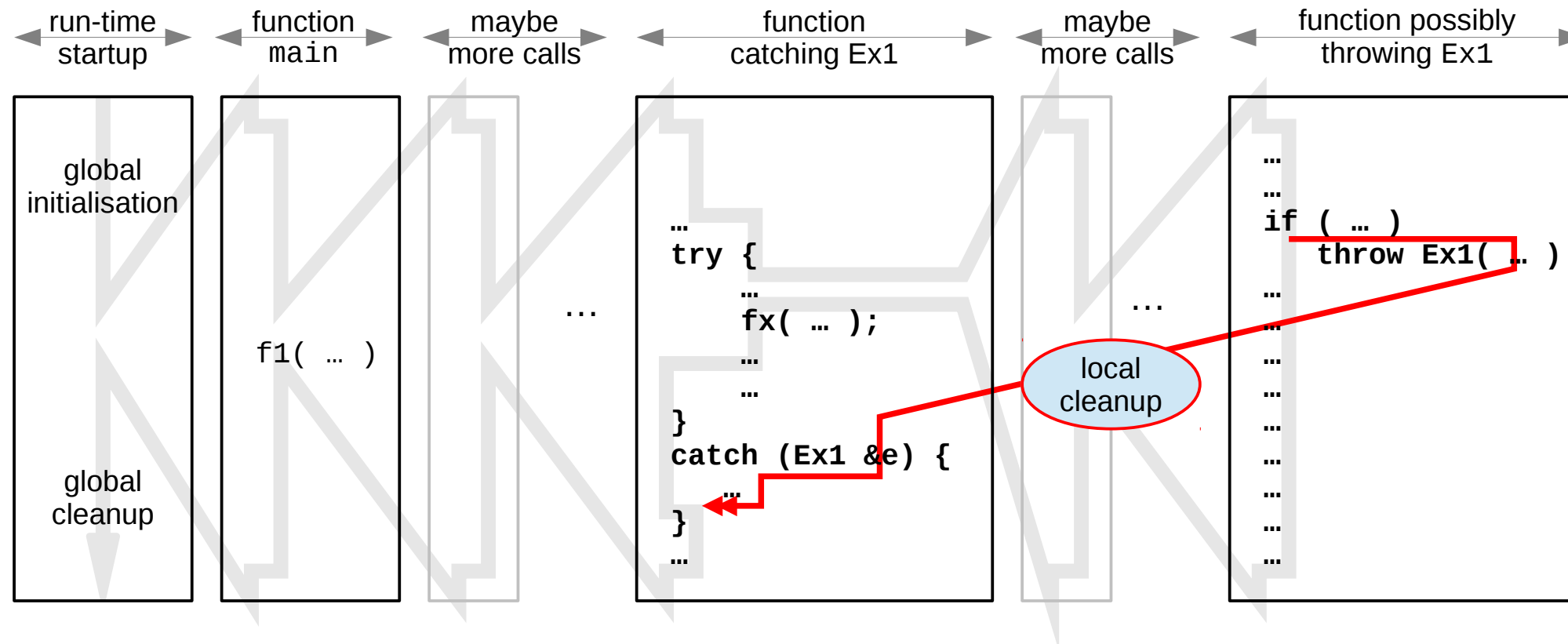
- types are typically deduced at the call site;
- may optionally be supplied, or
- must** be supplied if **not** present in parameter list (syntax then as for classes: concrete types in angle bracket list following identifier)

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

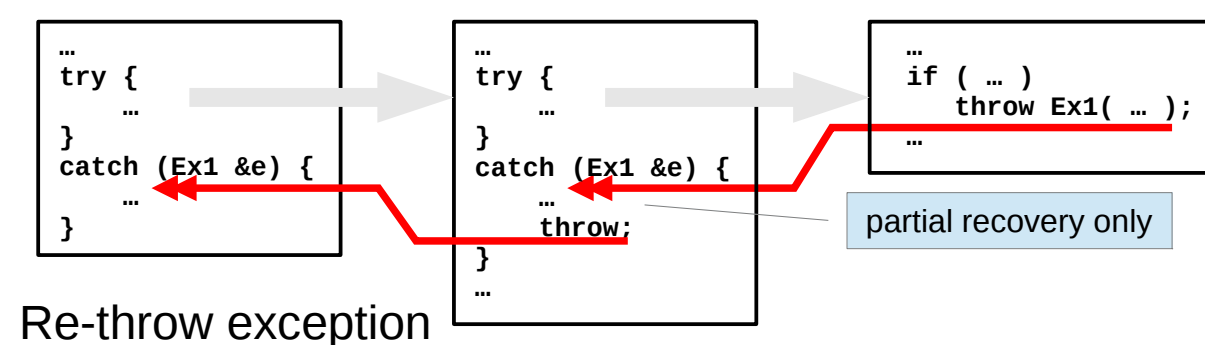
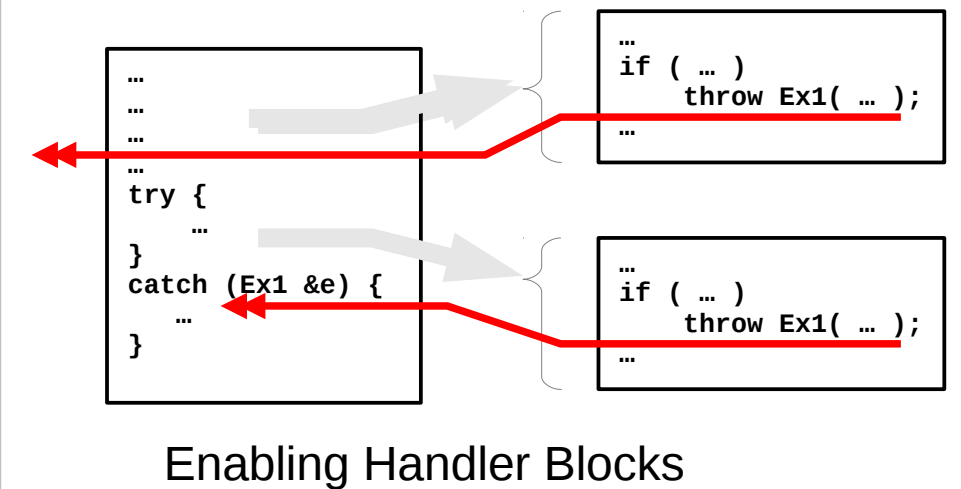
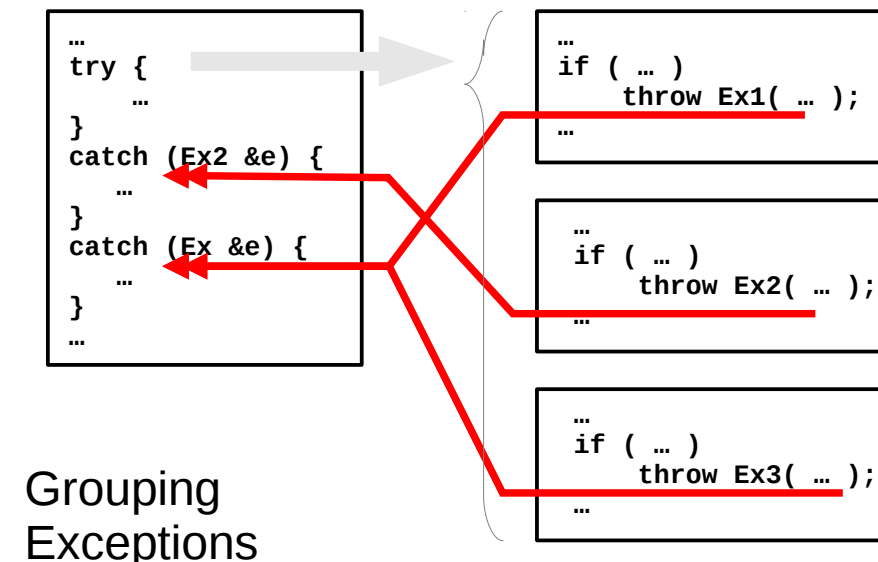
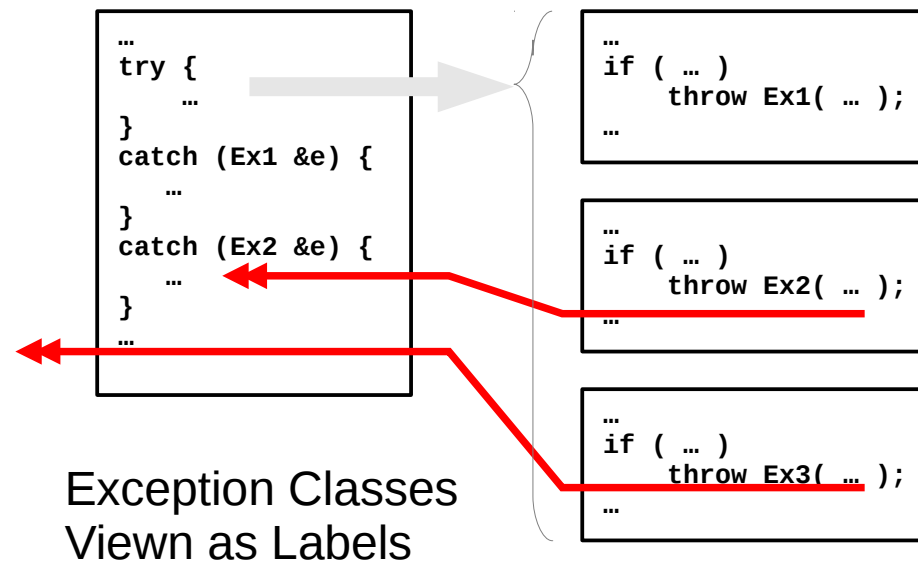
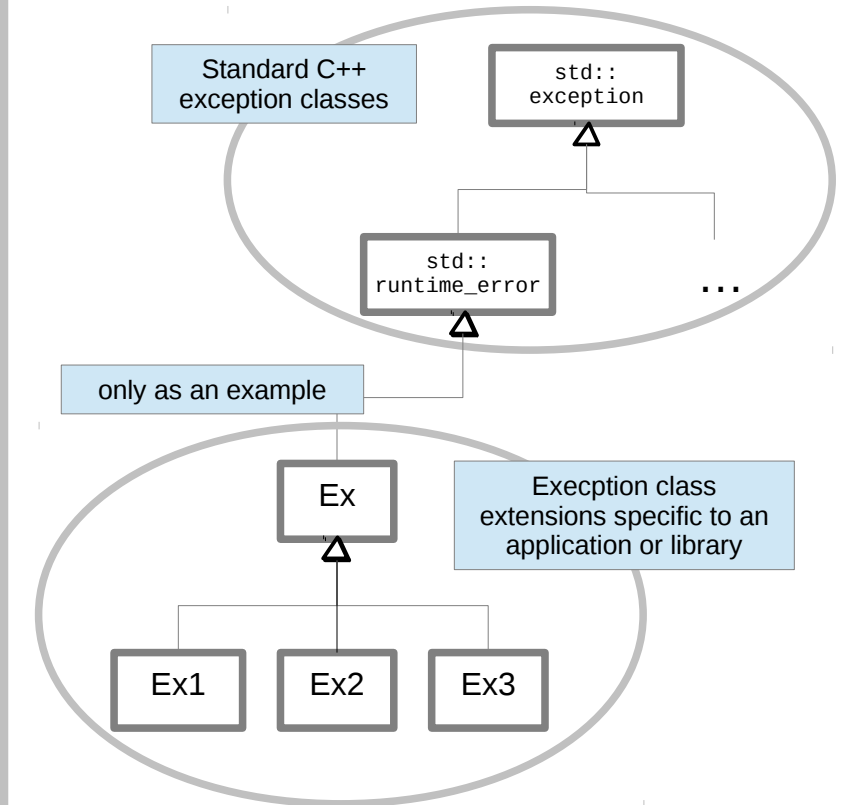
Code Compiled and Optimised for Specific Template Arguments

Template Basics

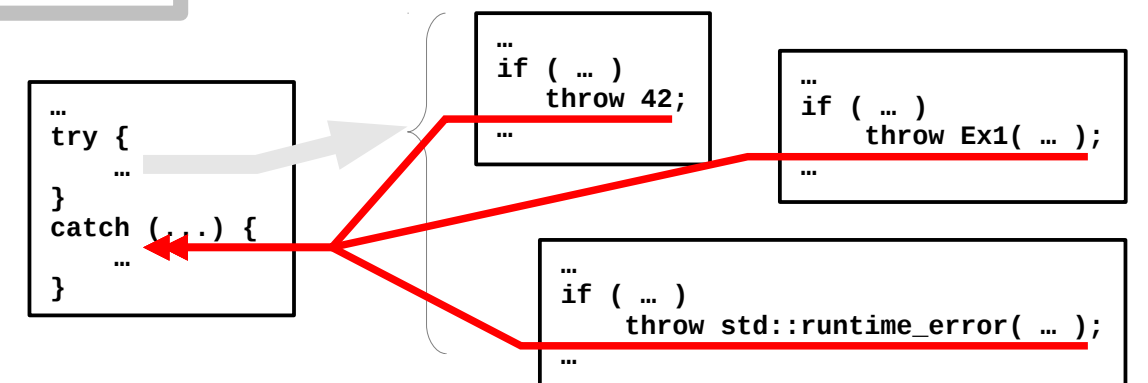
Execution Path taken for Exception



Exception Class Hierarchies

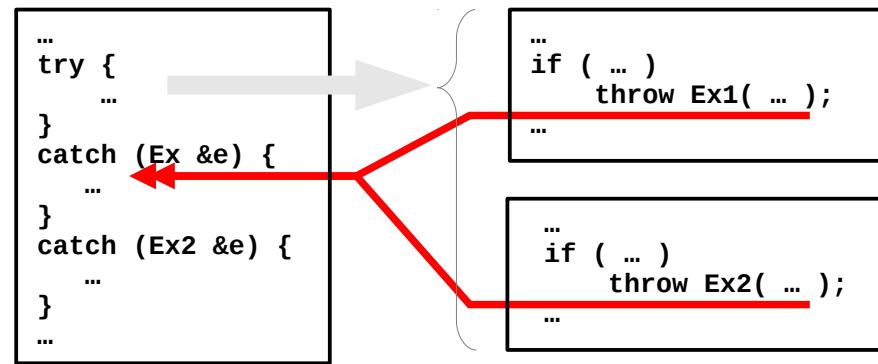


Catch Any Exception



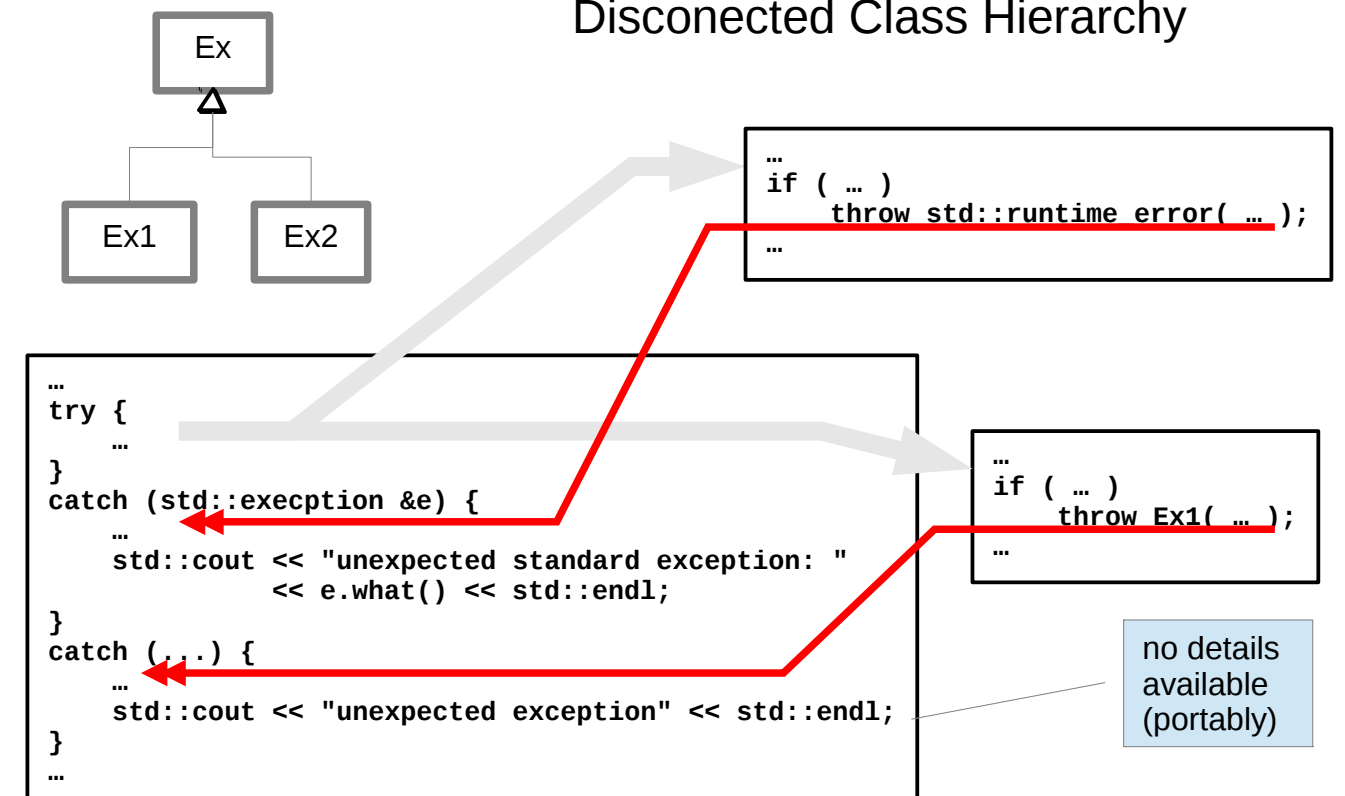
Exception Basics

Bad Order of Handlers

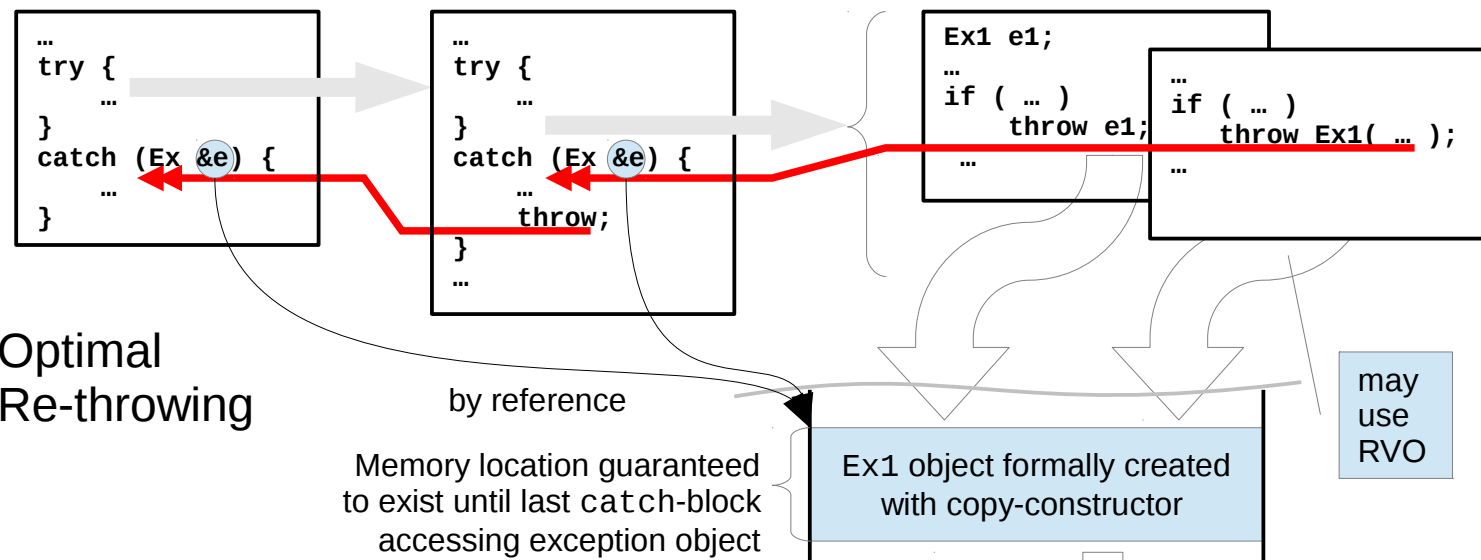


The compiler may issue a warning that the second catch-clause is shadowed by the first but this is not mandatory.

Disconnected Class Hierarchy



Optimal Re-throwing

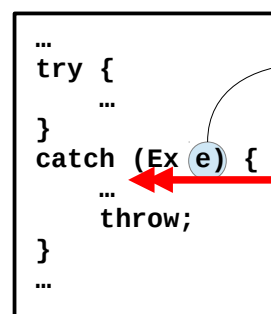
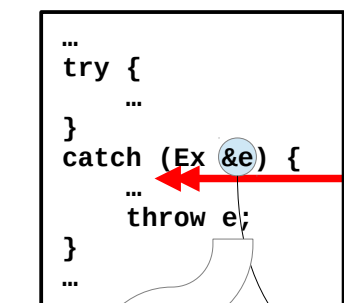


by reference

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

Ex1 object formally created with copy-constructor

may use RVO



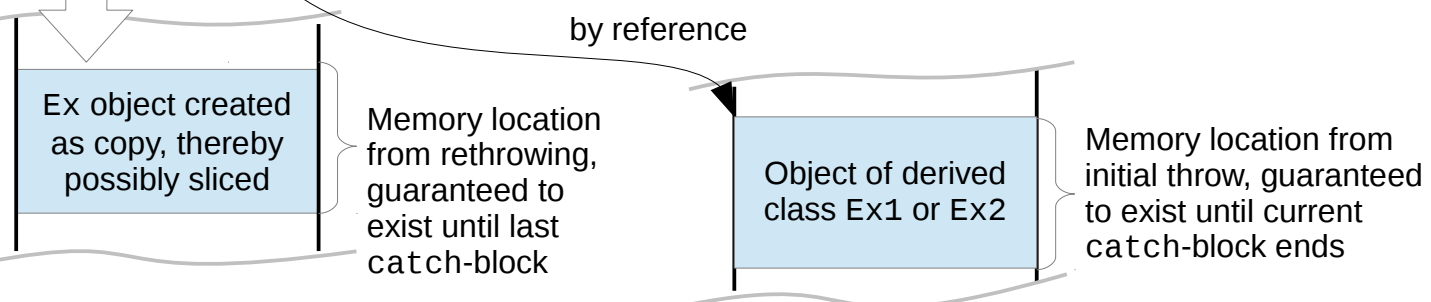
by value

Ex object created by slicing Ex1

physical copy, no polymorphism

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

Slicing Exception Object



by reference

Ex object created as copy, thereby possibly sliced

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

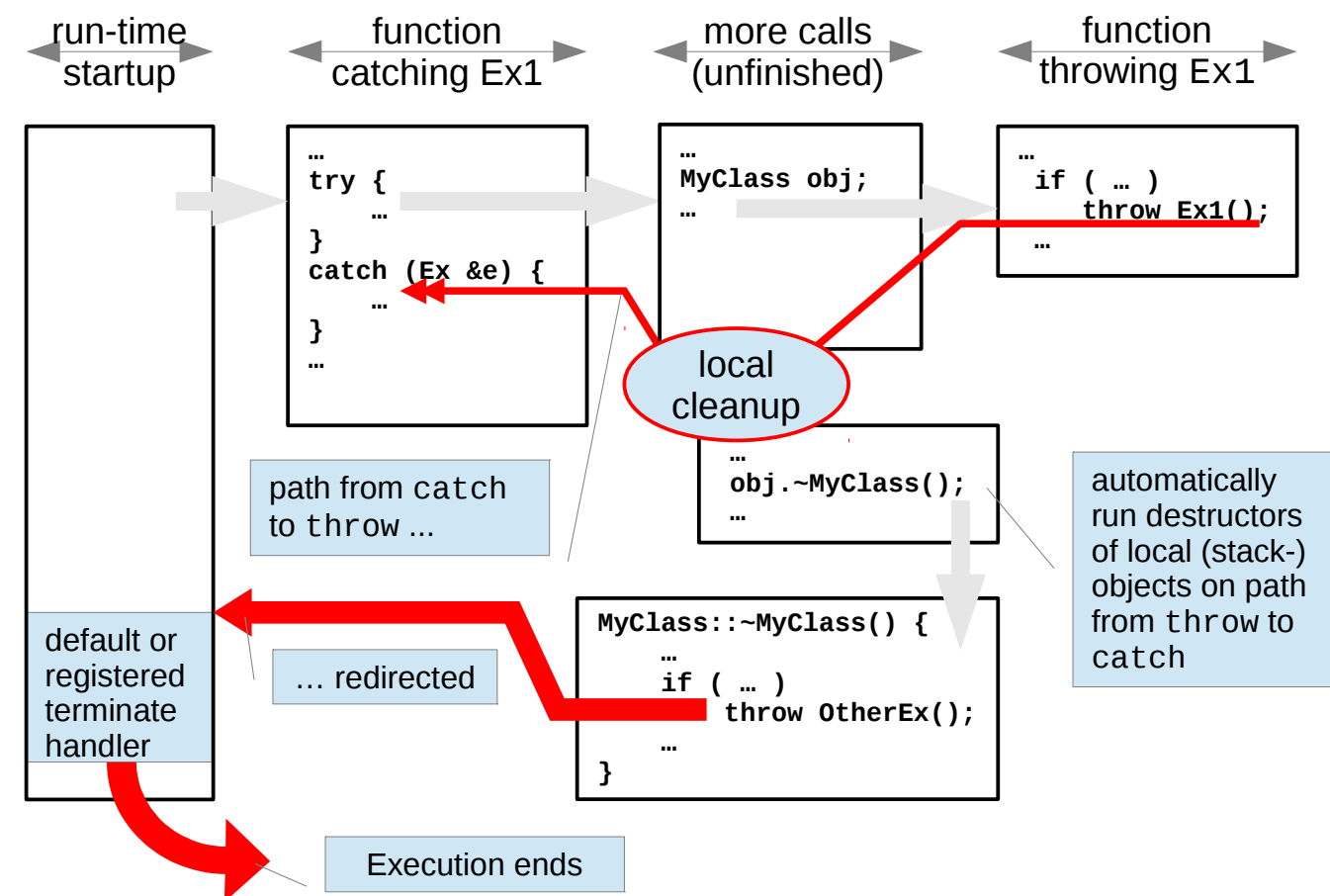
Object of derived class Ex1 or Ex2

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

Exception Details

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

Throwing from Destructors



path from catch to throw ...

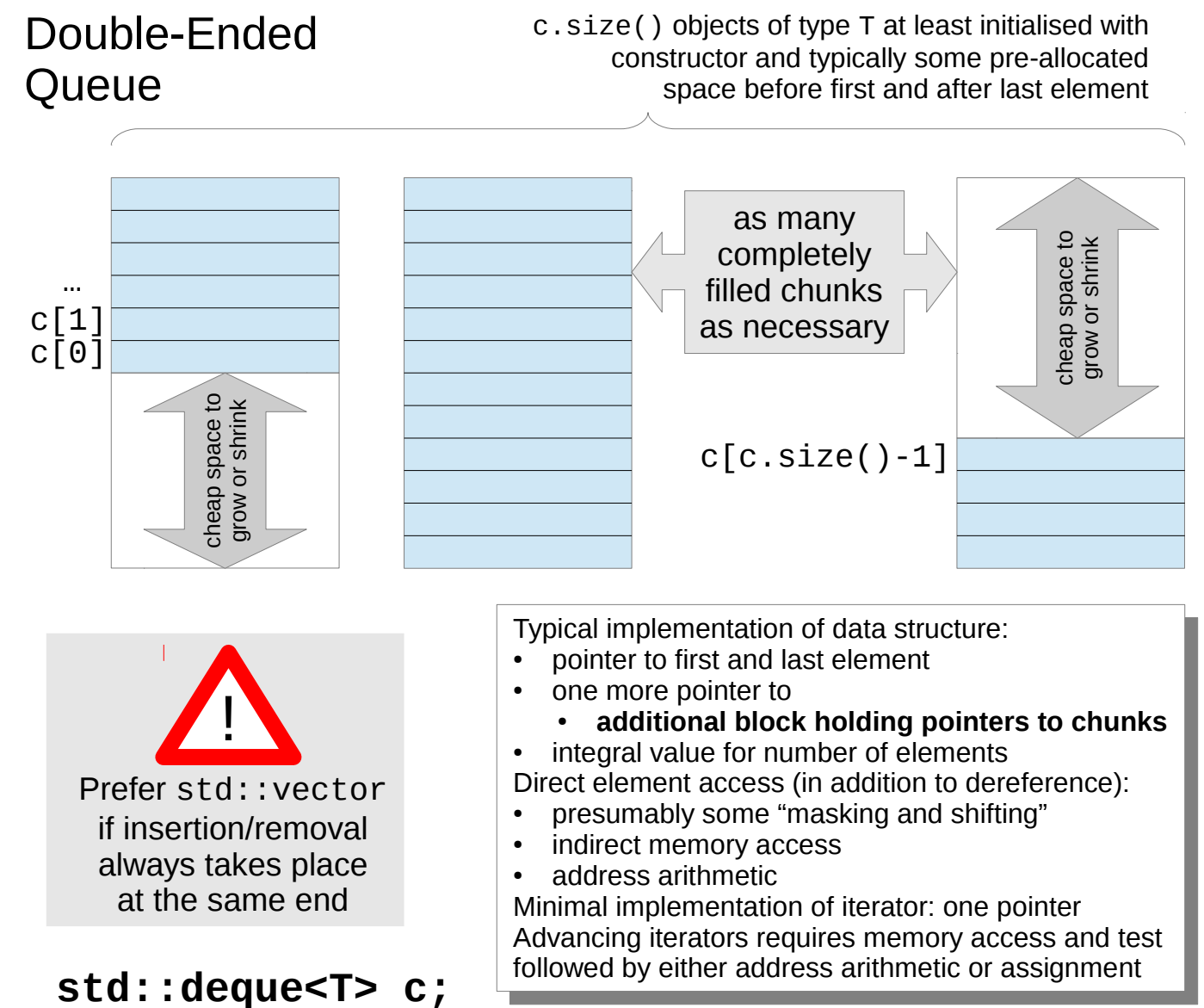
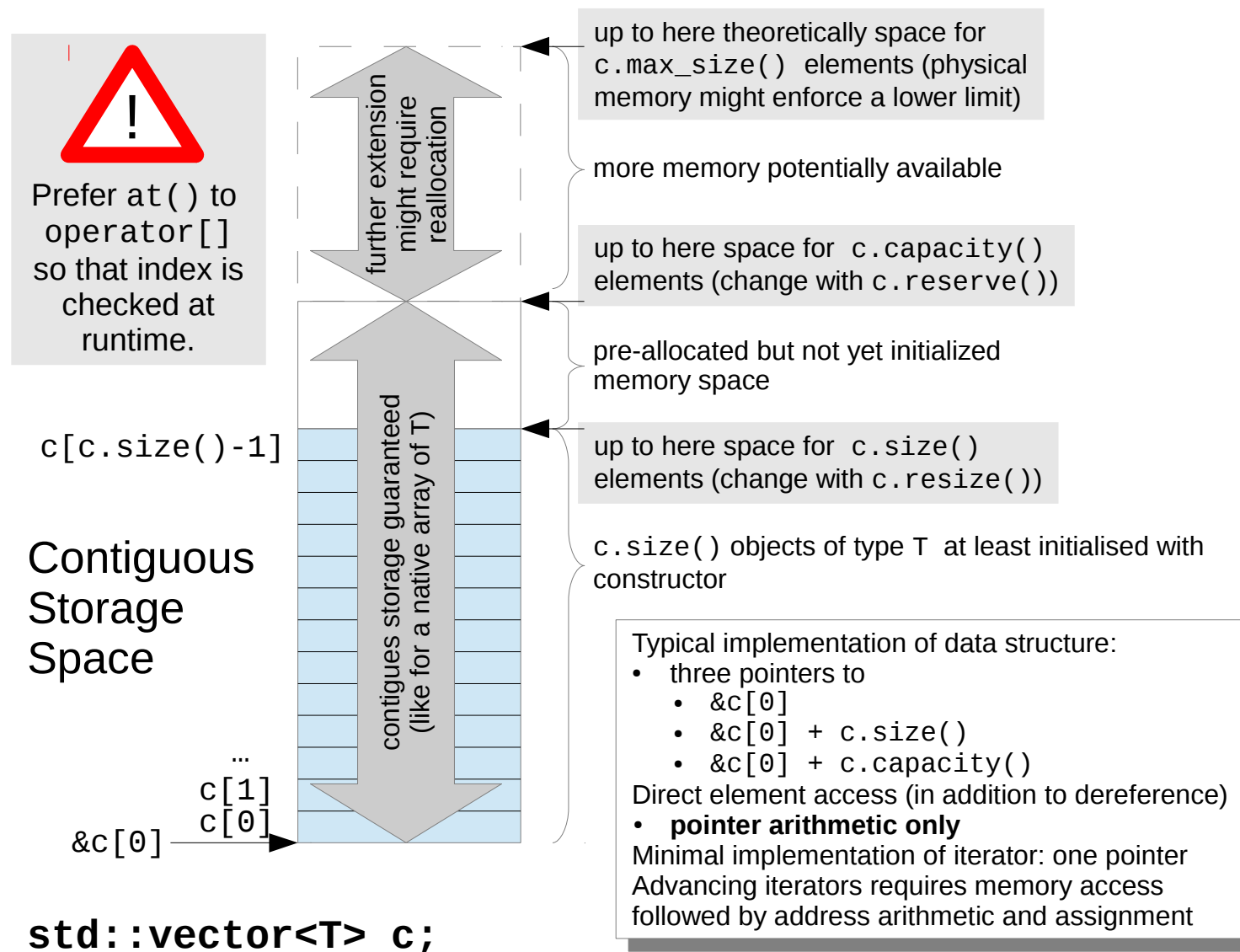
local cleanup

automatically run destructors of local (stack-) objects on path from throw to catch

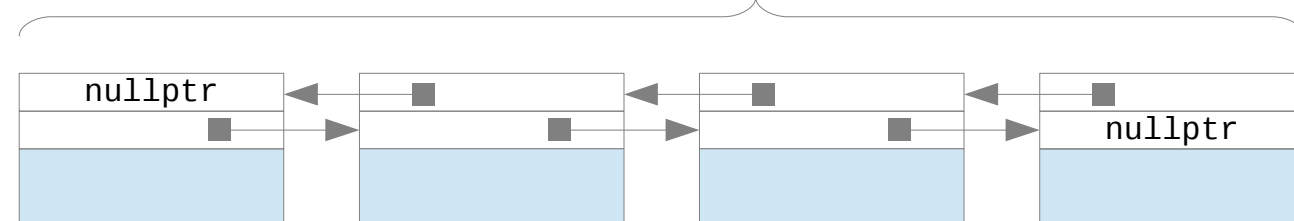
default or registered terminate handler

... redirected

Execution ends



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



Typical implementation:

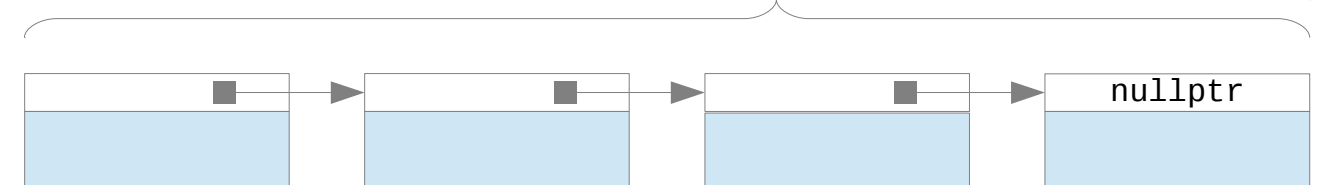
- **two pointers per element**
- pointer to first and last element
- integral value for number of elements

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

Double Linked List

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

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



Singly Linked List

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

Typical implementation:

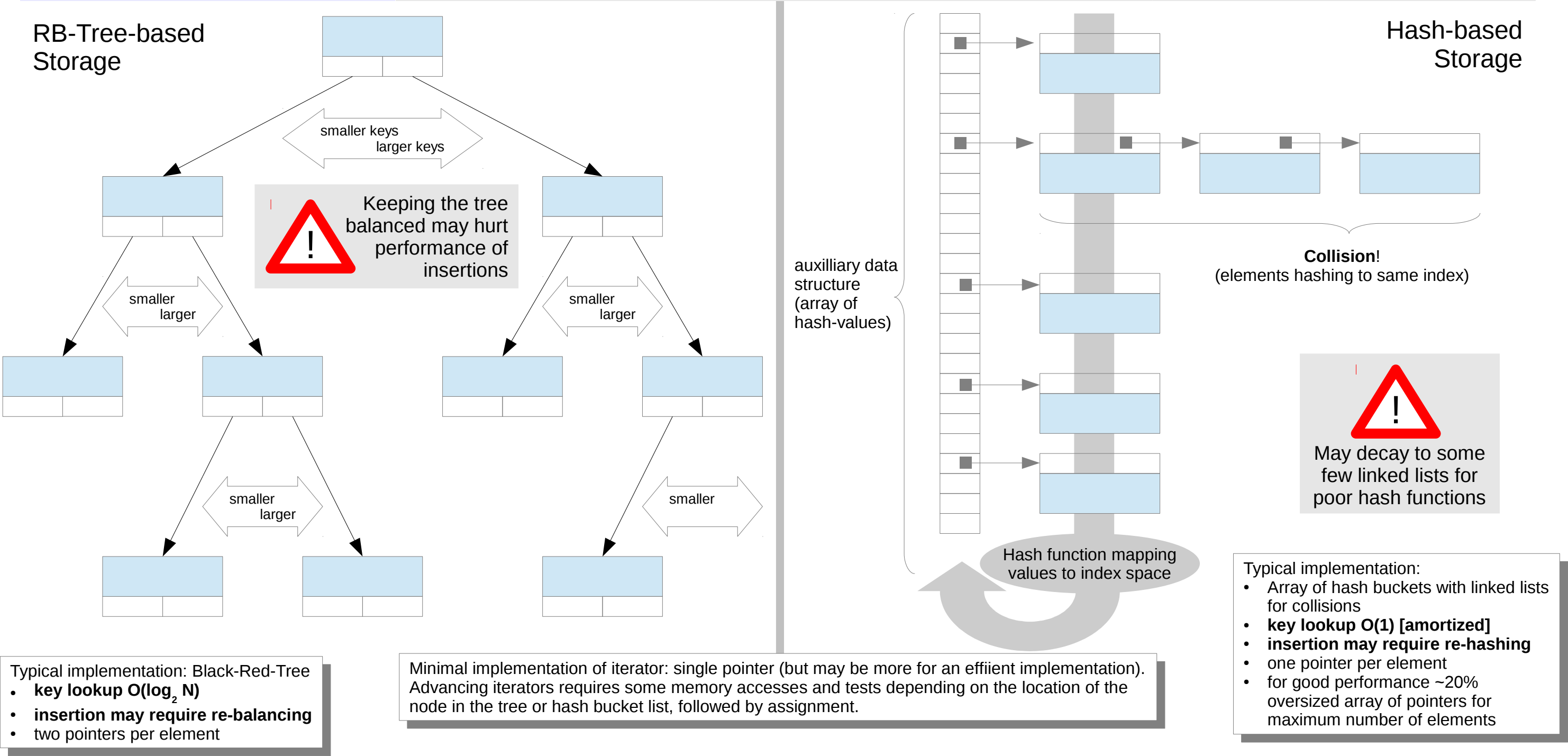
- **one pointer per element**
- only pointer to first element
- **number of elements not stored!**

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

STL – Sequence Container Classes

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

Contained elements	STL Class Name		Restrictions
objects of type T	<code>std::set</code>	<code>std::unordered_set</code>	unique elements guaranteed
	<code>std::multiset</code>	<code>std::unordered_multiset</code>	multiple elements possible (comparing equal to each other)
pairs of objects of type T_1 (key) and type T_2 (associated value)	<code>std::map</code>	<code>std::unordered_map</code>	unique keys guaranteed
	<code>std::multimap</code>	<code>std::unordered_multimap</code>	multiple keys possible (comparing equal to each other)



STL – Associative Container Classes

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

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

operations expected from iterators



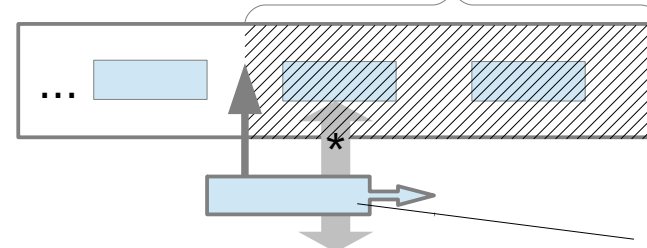
Failure to comply will cause a compile-time error, typically with respect to the header file that defines the algorithm.

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.

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



"Removing" Elements ... returns "New End"

Use of iterators to specify container elements to process:

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

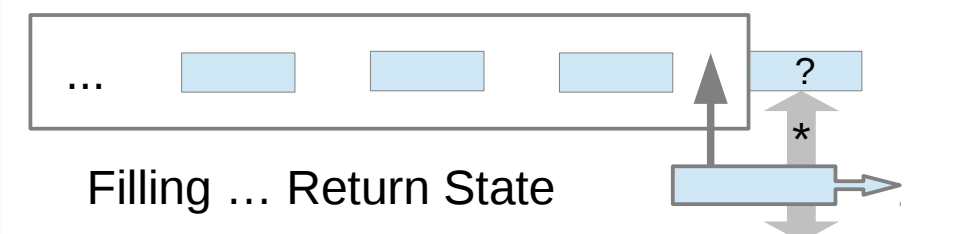


always valid for dereferencing

... Return Success ...

not necessarily valid for dereferencing!

... or Failure



Input Iterators Semantic Restrictions

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

Output Iterators Semantic Restrictions

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

STL – Iterator Usages

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 not 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	...
...	[\]	...	ä	...	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 mainly 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)

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



- 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

C-Compatibility

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

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

`CharType`
`std::basic_string`

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

`char`
`std::basic_string`

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

`wchar_t`
`std::basic_string` } `std::wstring`

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

... the other way round is automatically

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

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

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

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

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

Efficiency

hi!

a

b → Hello, world

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

a = b;

becomes (silently) **shared**

a

Hello, world

b

content copied on write ...

a[7] = 'W';

... and again **unshared**

a

Hello, World

Classes

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.

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

```
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 ...
```

convert to every builtin **integral** type

convert to every builtin **floating point** type

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

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

```
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) ...
...
```

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

C++11

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

Advanced operations:

- too many to list (→ RTFM).
- Process like an STL container:**
- the usual iterator subclasses are defined as for containers;
- therefore all STL algorithms may be used on `std::string` too.

C++11 has added some “missing” ones:

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

May or May Not Have COW-Implementation

Basic String Operations

Similar to builtin types:

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

Similar to builtin arrays:

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

As in some other programming languages:

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

Basic searching:

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

Partition:

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

Numeric Conversions

Standard Strings 101

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

Advanced String Operations

Basics



Substantial overhead if
sizeof(T) is small.

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;

...
int main() {
    regex re("he(1+)(o*)");
    ...
    ...
    string line;
    while (getline(cin, line)) {
        ... re ... // execute FSM
    }
}
```

Regular Expression
represented in Text Form

constructor

Regex-Object
(FSM representing the RE)

Regular
Expression
Object

Substitution Format:

- may contain any text
- plus the placeholders
\$0, \$1, ... for parts of
the compared string
matching parts of the
regular expression
put in parentheses.

```
...
const char fmt[] = R("
-----
complete match: $0
matching el-s:  $1
matching o-s:   $2
-----
");
...
regex_replace(line, re, fmt) ...
...
```

```
-----
complete match: helloo
matching el-s:  ll
matching o-s:   oo
-----
```

Match-Object:

- allows to access the parts of a string
matching the parts of a regular
expression put into round parentheses;
- has also a size() member function.

```
...
smatch m;
if (regex_search(line, m, re)) {
    // access matching parts
    ... m[0] ...
    ... m[1] ...
    ... m[2] ...
}
...
```

s	a	y		h	e	l	l	o	o	\n
---	---	---	--	---	---	---	---	---	---	----

Eingabezeile

Flexible Comparisons

```
...
string line;
while (getline(cin, line)) {
    if (regex_match(line, re)) {
        // line FULLY matches RE
        // ... whatever
        // ...
    }
    else {
        // no full match
        // ...
    }
}
...
```

```
...
string line;
while (getline(cin, line)) {
    if (regex_search(line, re)) {
        // some PART of line matches RE
        // ... whatever
        // ...
    }
    else {
        // no partial match
        // ...
    }
}
...
```

hel hell hellll helllll helo heloo helloo
say hello hello? say hello! say hello, oh?!
heo say hello Othello
say Hello Goodbye

Powerful Substitutions

Regular Expressions

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

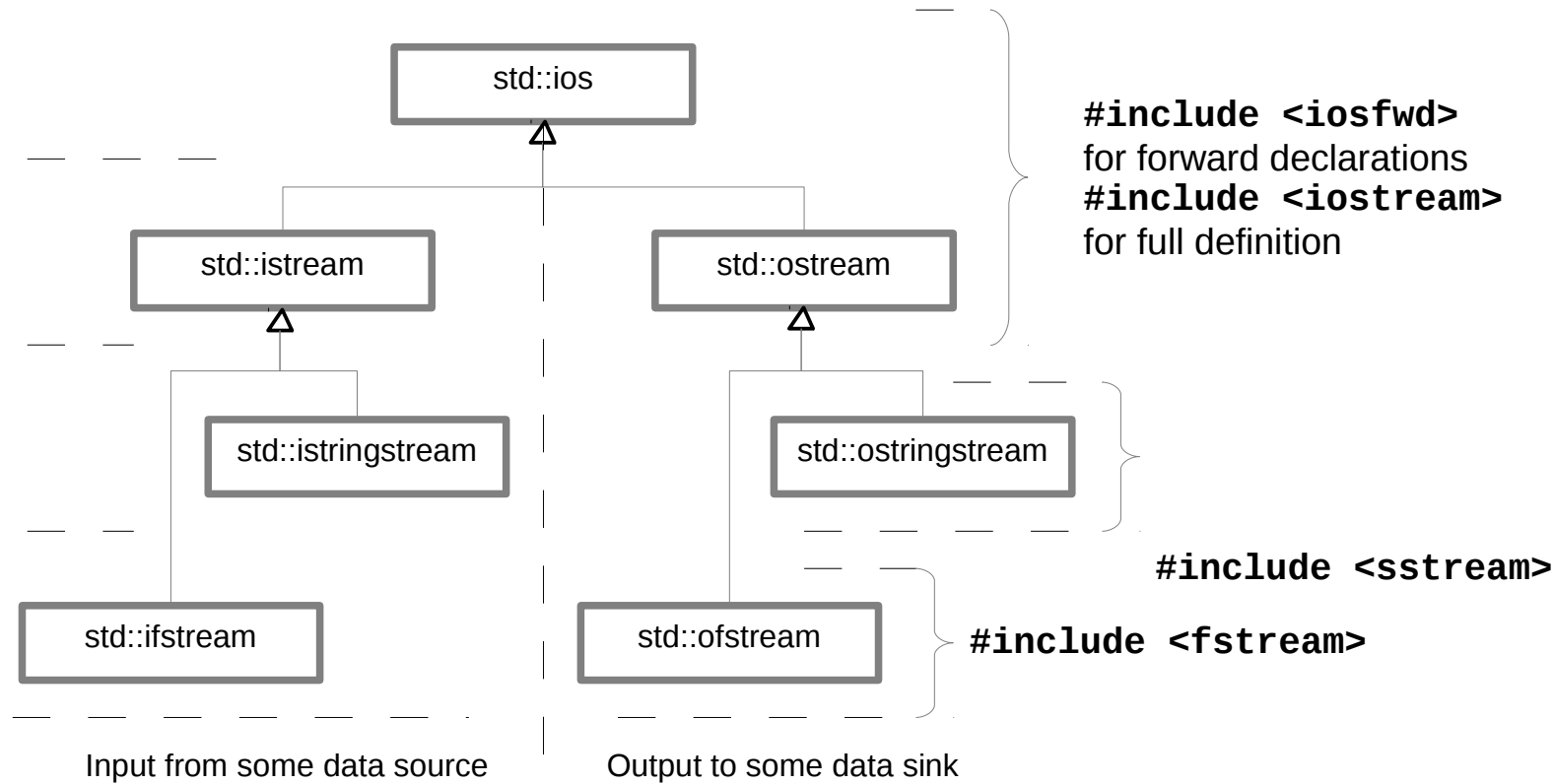
Easy Parsings

Common type definitions, constants, etc.

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

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

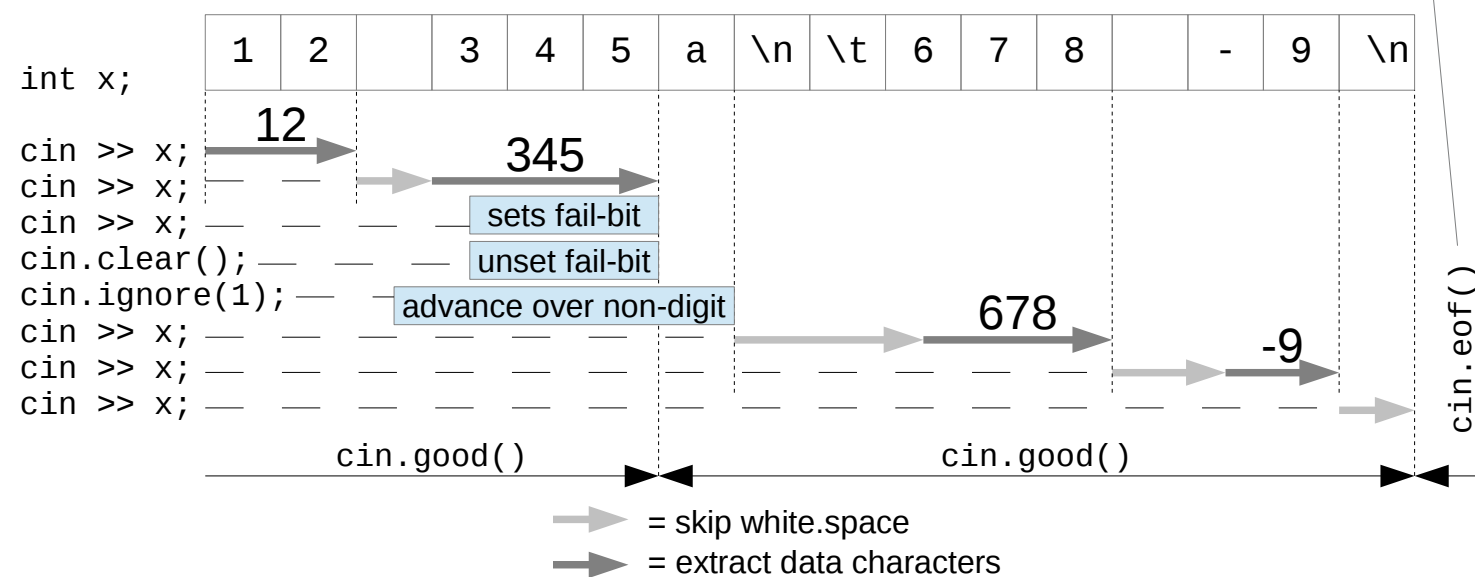
I/O to/from external sources and sink (typically classic files or devices)



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

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

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



```
used in standard library
for implementation of
std::istream
std::ostream
std::ifstream
std::ofstream
```

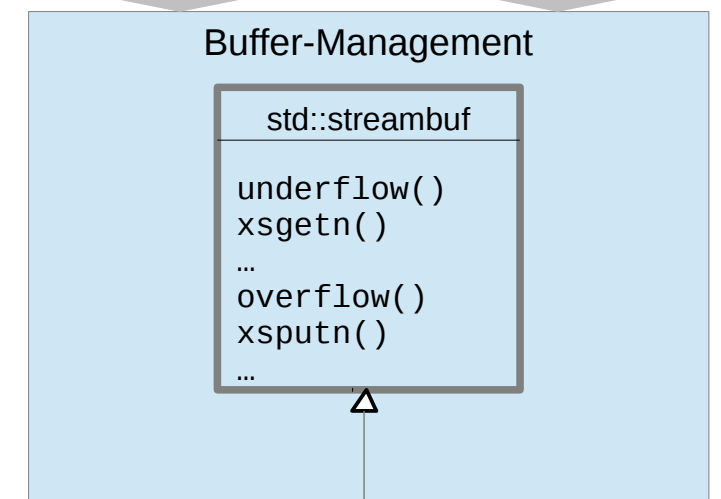
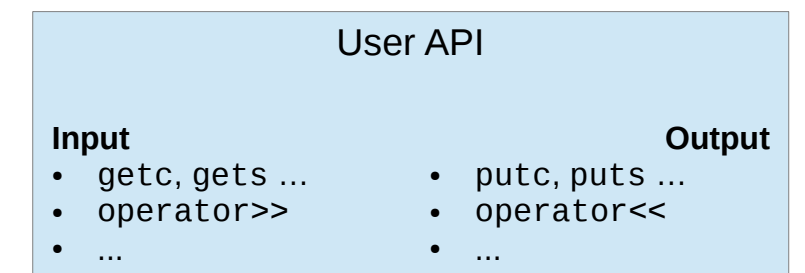
useful for individual
extensions though
special knowledge
must be acquired

I/O-Streams State-Bits

I/O-Stream Basics

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

“day to day” use of C++



available for in-memory
I/O with `std::string-s`
and classic files/devices

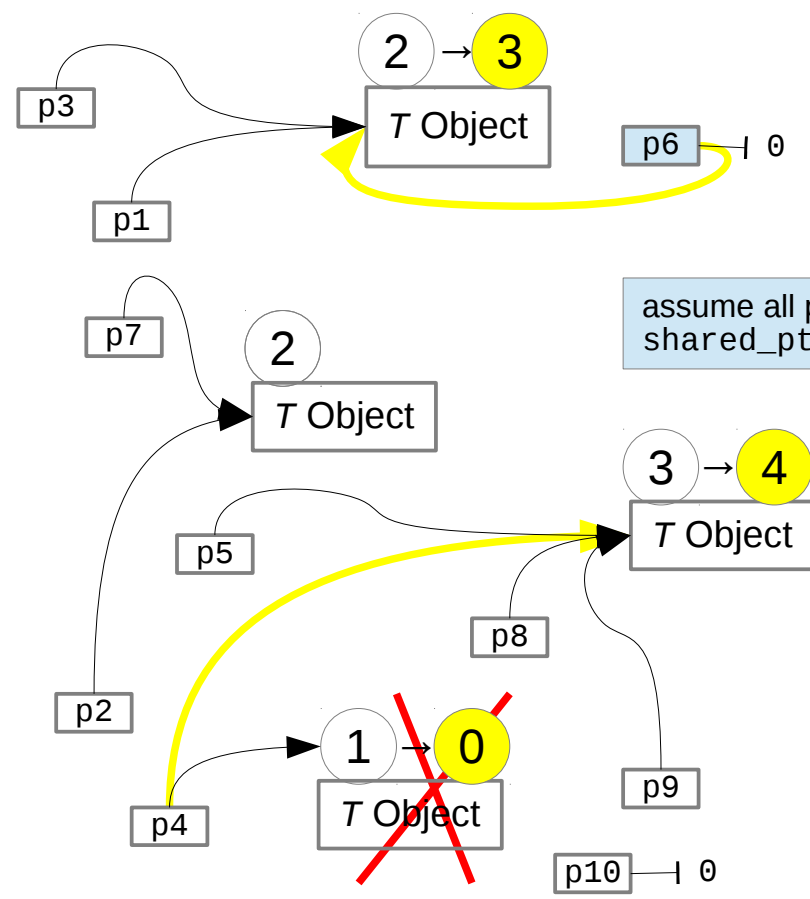
Mandatory overrides:

- underflow for input (provide one more character when buffer is exhausted)
- overflow for output (extract one character when buffer is full)

More overrides may improve performance:

- `xsgn` (provide more than one character)
- `xspn` (extract more than one character)
- ...

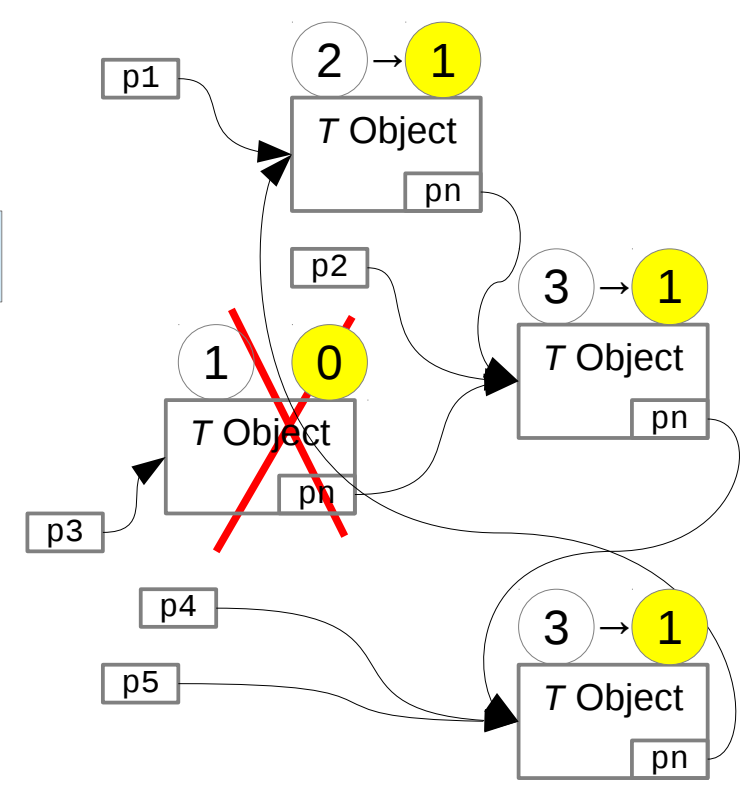
I/O-Streams Back-End



assume all pointers
shared_ptr<T>

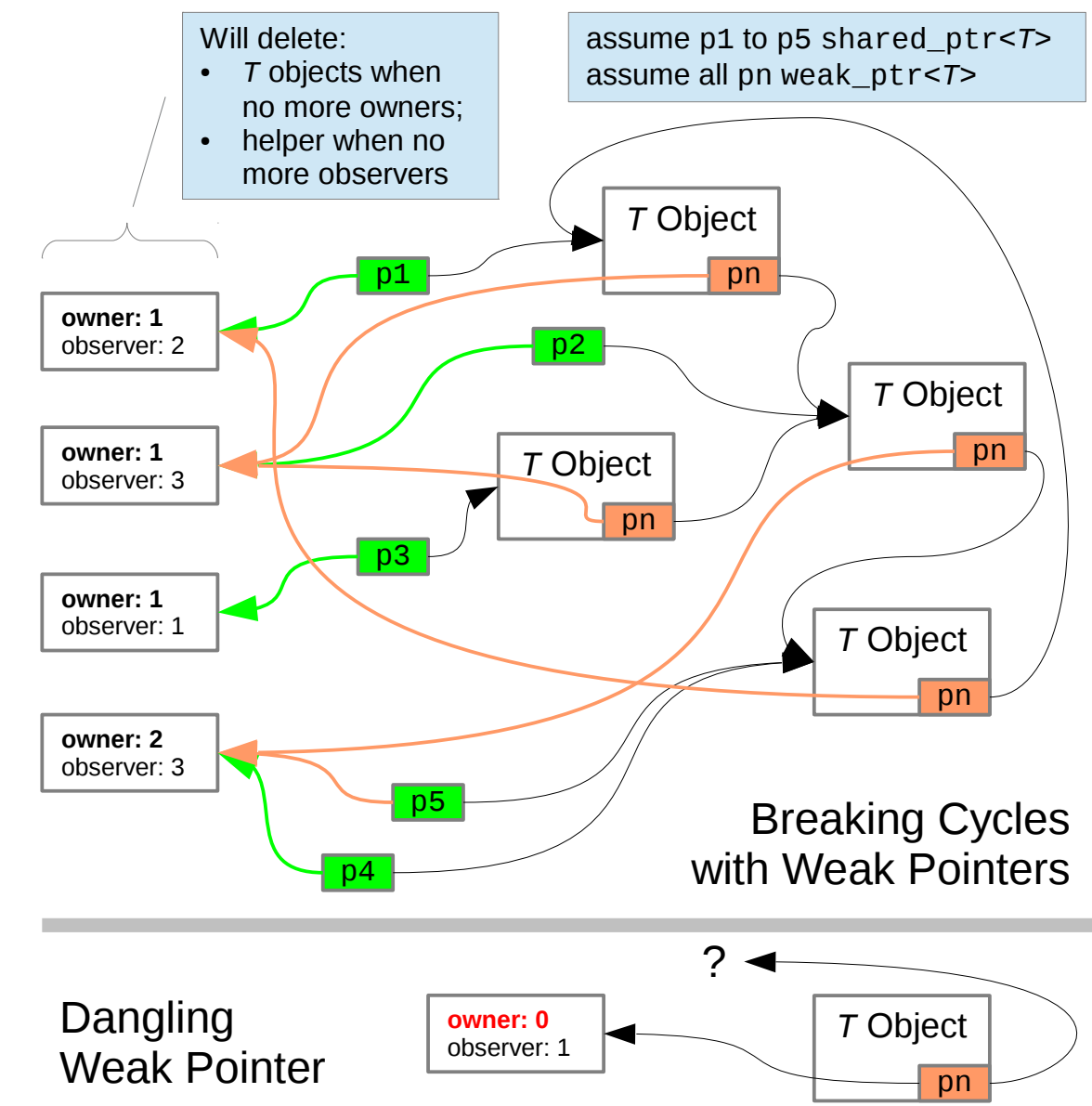
```
// assume assignments
p6 = p3;
p4 = p5;
```

Reference Counting Principle



```
// assume life-time
// of p1 to p5 ends
```

Problem of Cyclic References



Will delete:

- T objects when no more owners;
- helper when no more observers

assume p1 to p5 shared_ptr<T>
assume all pn weak_ptr<T>

Breaking Cycles with Weak Pointers

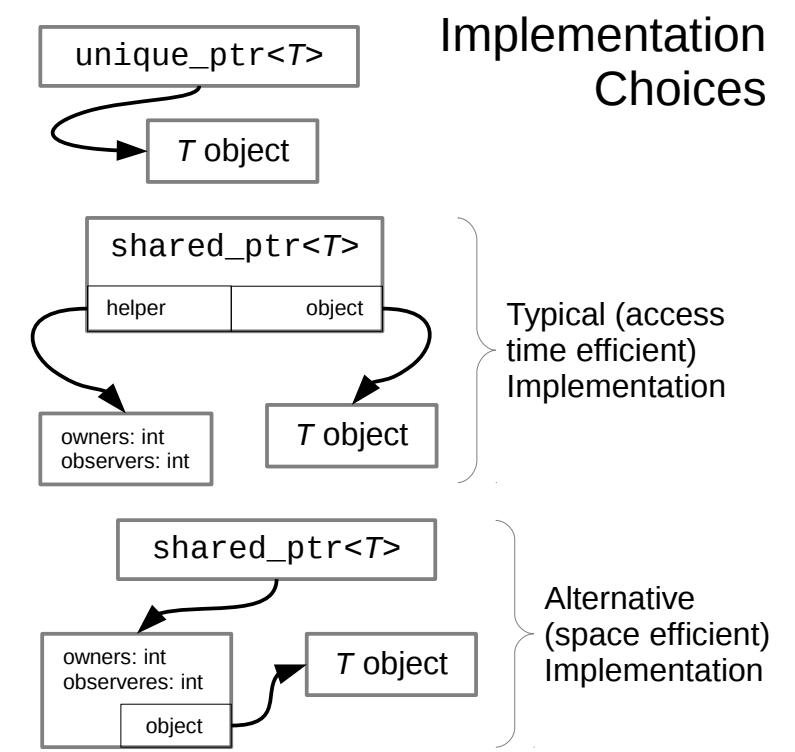
Dangling Weak Pointer

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*		
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	a T destructor must also be called in an assignment if the current referrer is the only one referring to the object

*: explicit use of std::move for argument is possible

Smart Pointer Comparison

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



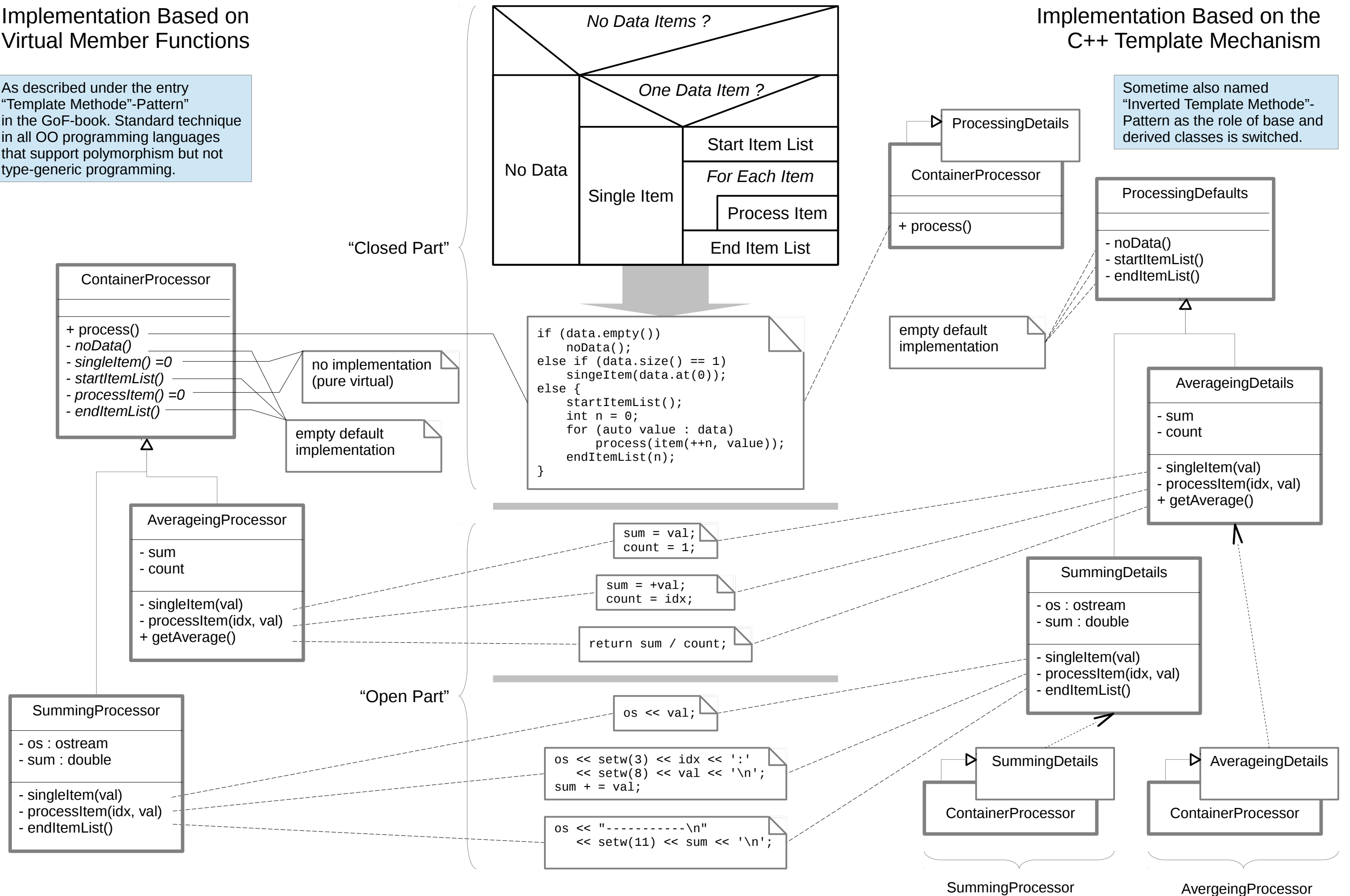
Implementation Choices

Typical (access time efficient) Implementation

Alternative (space efficient) Implementation

Implementation Based on Virtual Member Functions

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



Example – “Open Close”-Principle

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