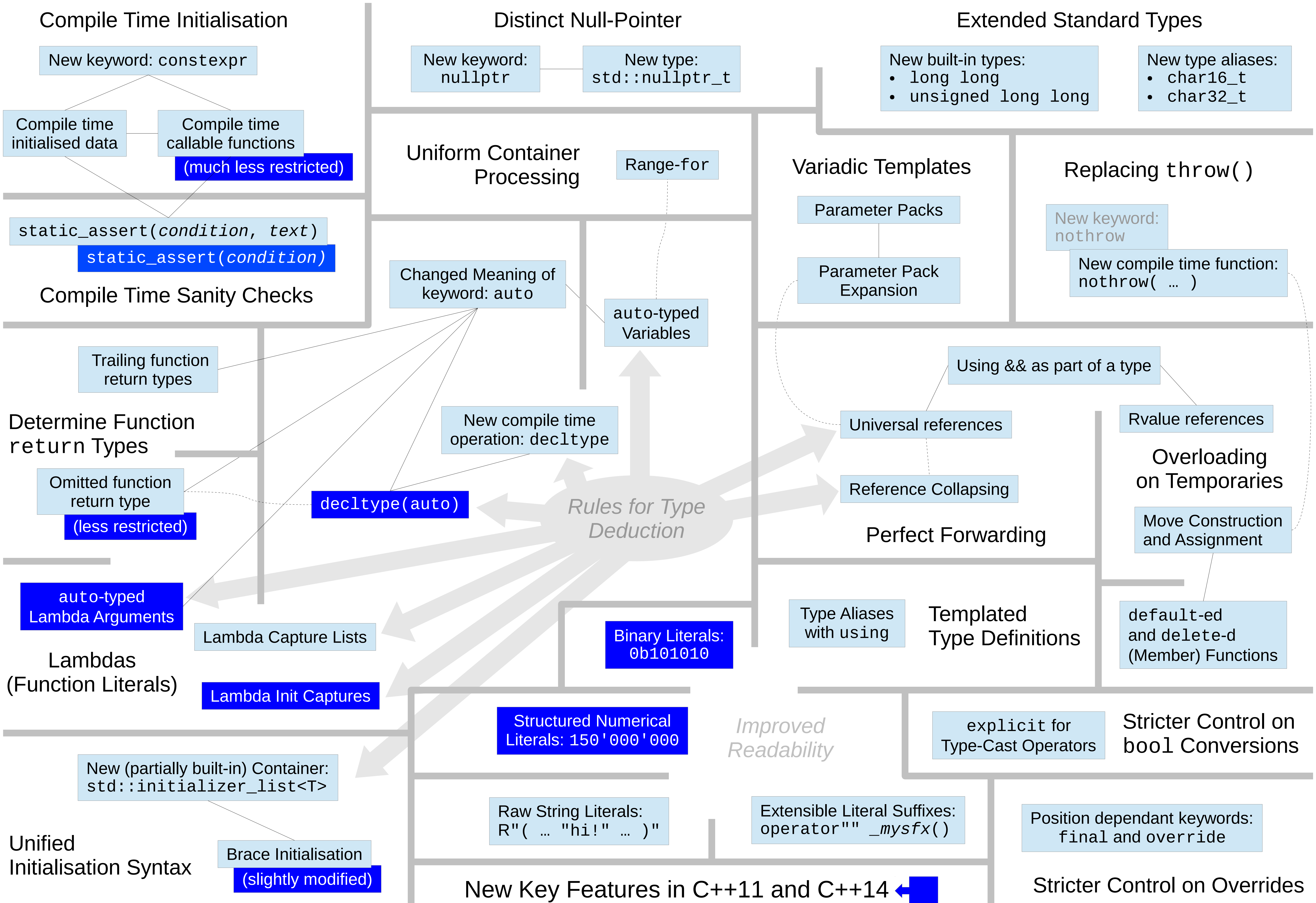


Compile Time Initialisation
New keyword: constexpr
Compile time initialised data
Compile time callable functions
(much less restricted)
static_assert(condition, text)
static_assert(condition)
Compile Time Sanity Checks
Distinct Null-Pointer
New keyword: nullptr
New type: std::nullptr_t
Extended Standard Types
New built-in types:
• long long
• unsigned long long
New type aliases:
• char16_t
• char32_t
Uniform Container Processing
Range-for
Variadic Templates
Parameter Packs
Parameter Pack Expansion
Replacing throw()
New keyword: nothrow
New compile time function: nothrow(…)
Changed Meaning of keyword: auto
auto-typed Variables
Trailing function return types
Determine Function return Types
New compile time operation: decltype
Omitted function return type
(less restricted)
decltype(auto)
Rules for Type Deduction
Using && as part of a type
Universal references
Rvalue references
Reference Collapsing
Overloading on Temporaries
Perfect Forwarding
Move Construction and Assignment
auto-typed Lambda Arguments
Lambda Capture Lists
Lambdas (Function Literals)
Lambda Init Captures
Binary Literals: 0b101010
Type Aliases with using
Templated Type Definitions
default-ed and delete-d (Member) Functions
Structured Numerical Literals: 150'000'000
Improved Readability
explicit for Type-Cast Operators
Stricter Control on bool Conversions
New (partially built-in) Container: std::initializer_list<T>
Unified Initialisation Syntax
Brace Initialisation
(slightly modified)
Raw String Literals: R"(… "hi!" …)"
Extensible Literal Suffixes: operator"" _mysfx()
Position dependant keywords: final and override
New Key Features in C++11 and C++14
Stricter Control on Overrides

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)

]

¤

€

no precomposed form

e

Ü

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

!

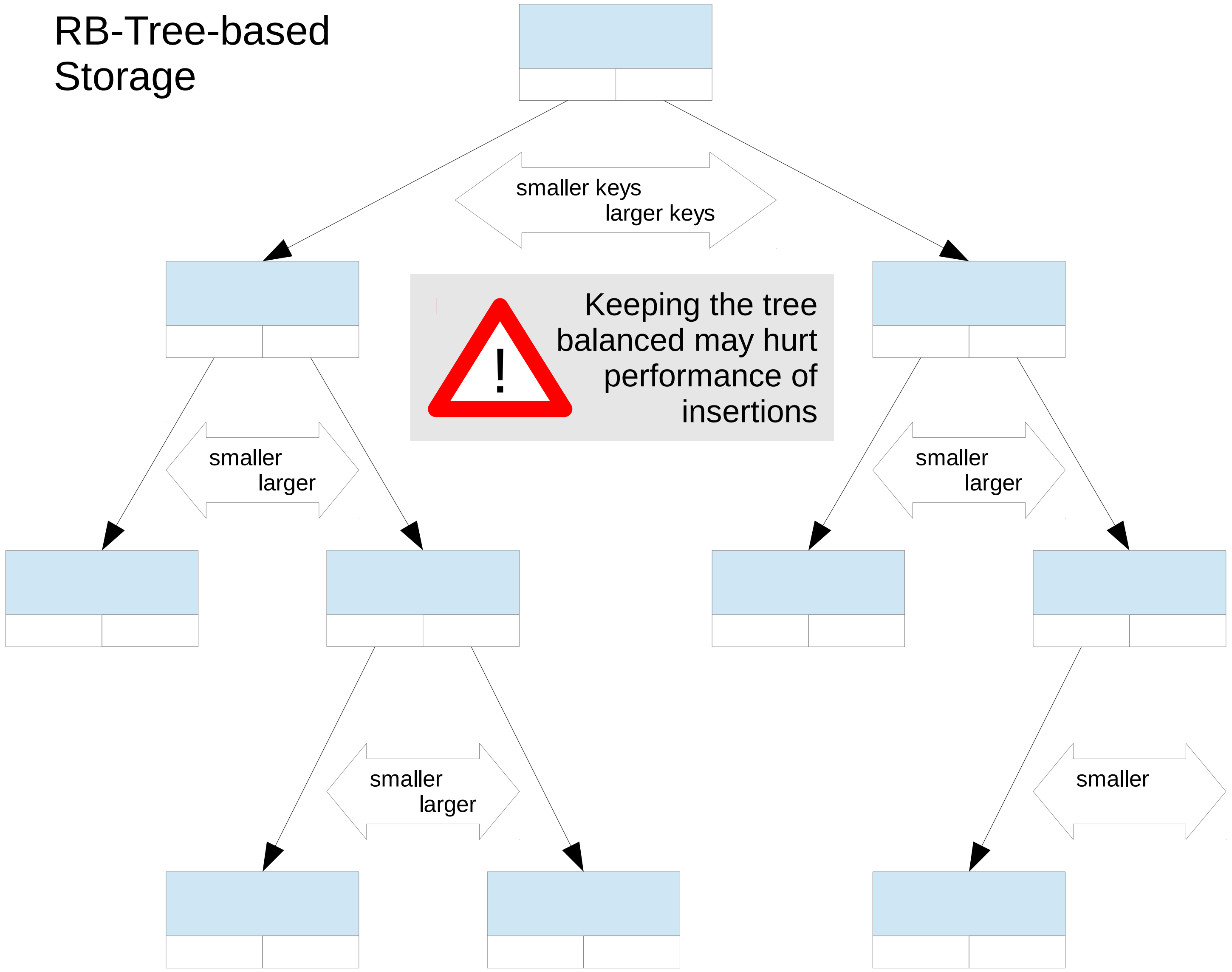
- C++ character and string types are templated on the code unit size.
- No information about the **character set** is carried in the type!
- Furthermore, type `wchar_t` is implementation defined.

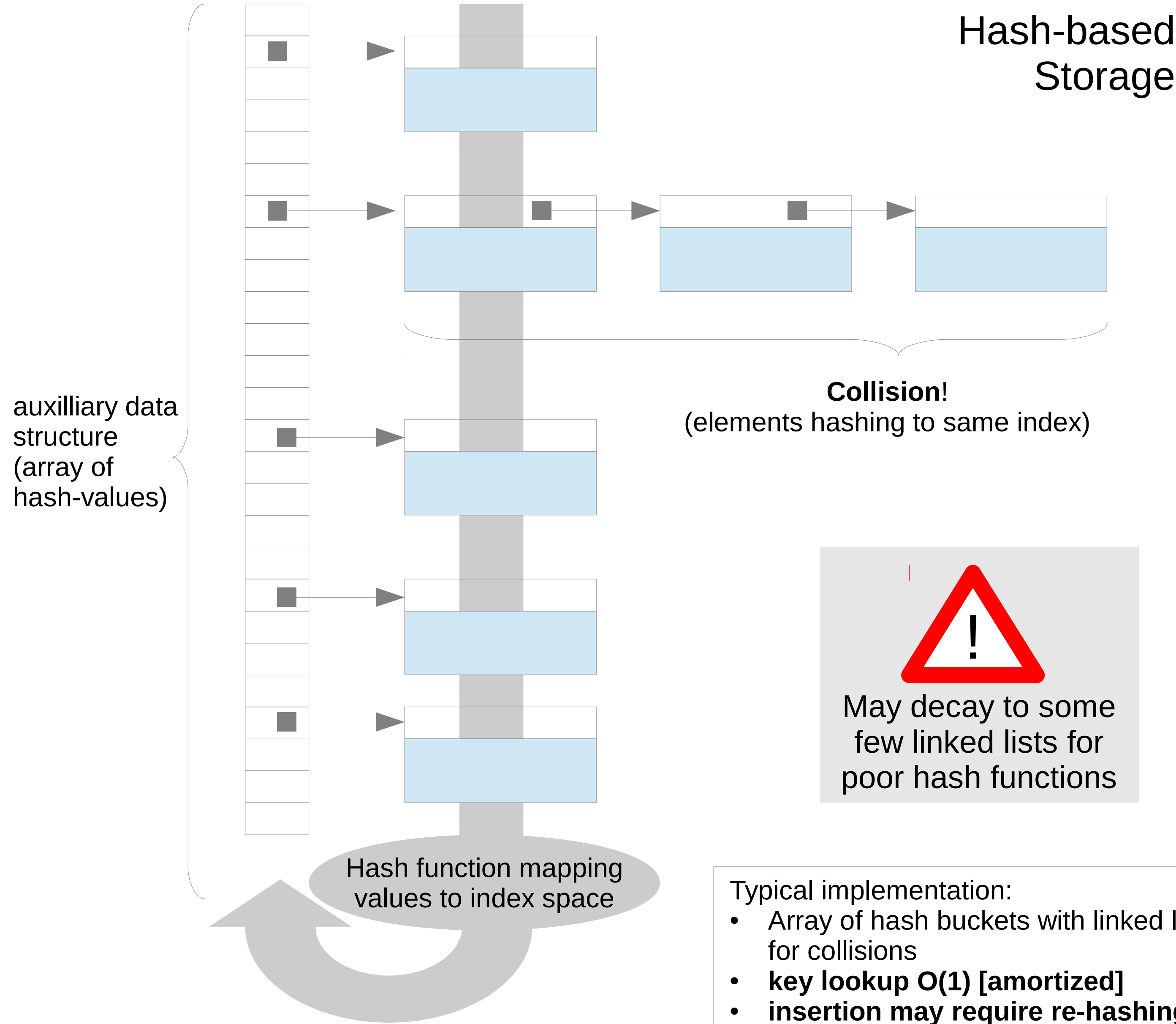
!

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

Code-Units vs. Code-Points

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)



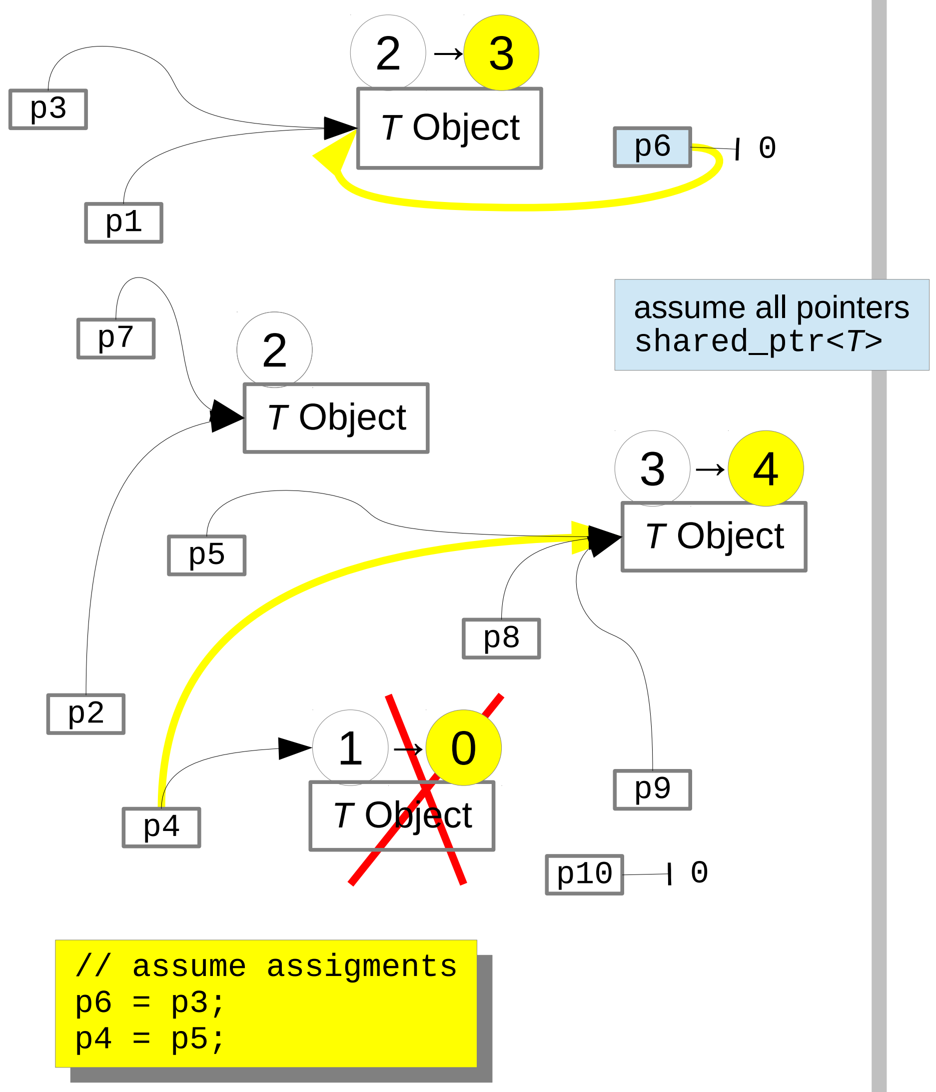


Typical implementation: Black-Red-Tree
- **key lookup O($\log_2$ N)**
- **insertion may require re-balancing**
- two pointers per element

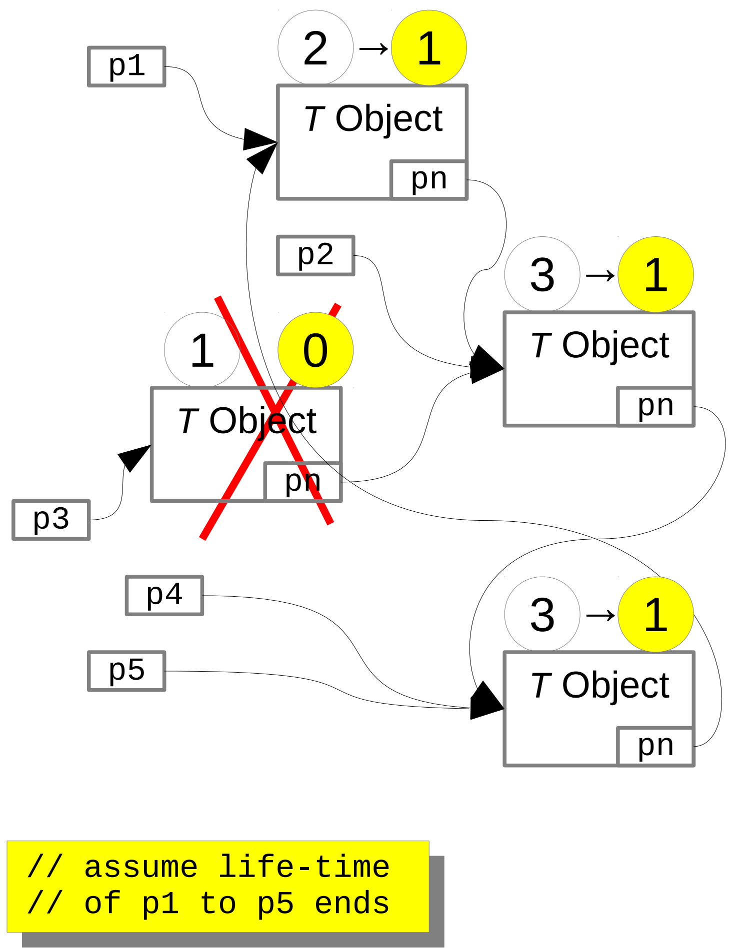
Minimal implementation of iterator: single pointer (but may be more for an effiient implementation). Advancing iterators requires some memory accesses and tests depending on the location of the node in the tree or hash bucket list, followed by assignment.

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

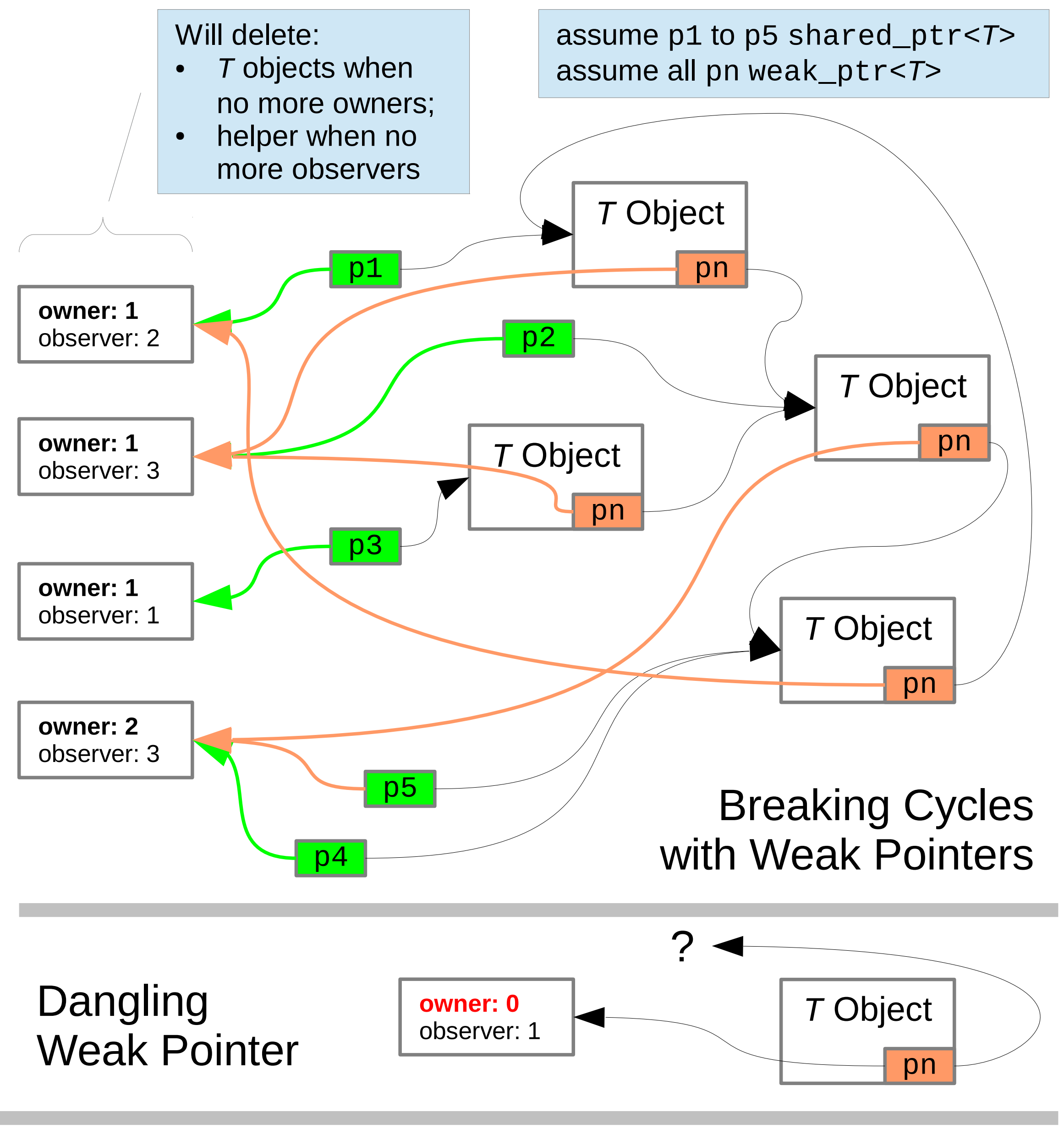
STL – Associative Container Classes



Reference Counting Principle



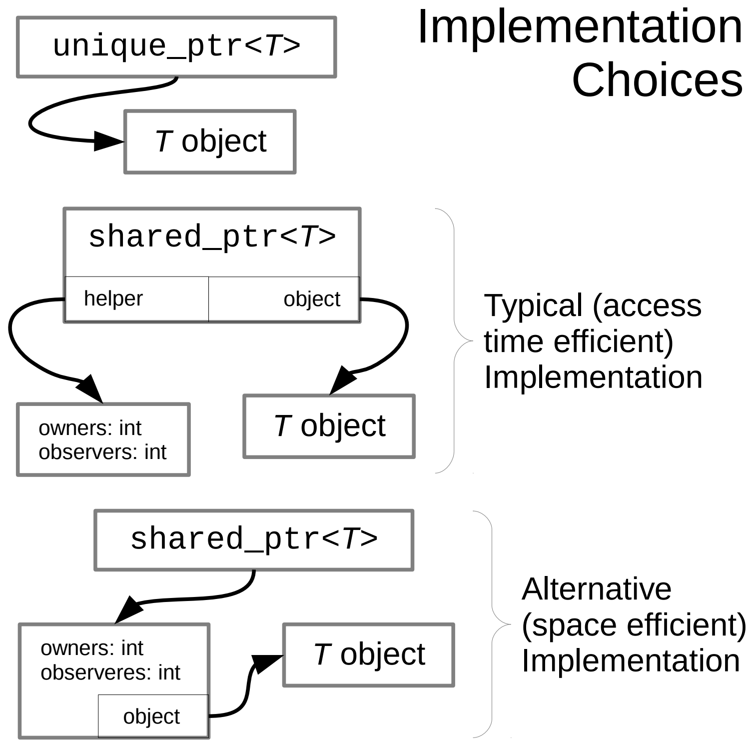
Problem of Cyclic References



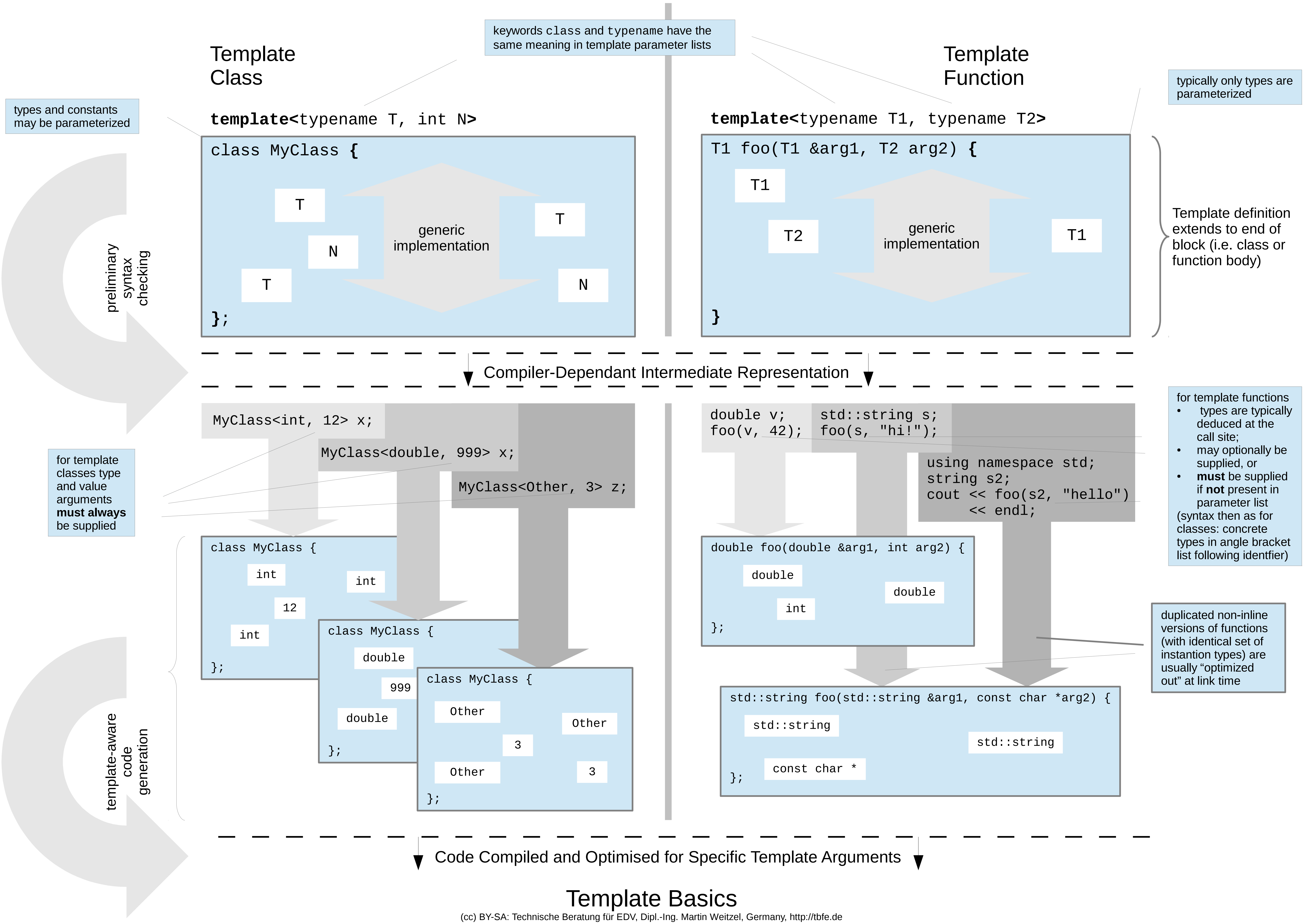
Comparing ...	`std::unique_ptr<T>`	`std::shared_ptr<T>`	**Remarks**
Characteristic	refers to a single object of type *T*, **uniquely owned**	refers to a single object of type *T*, **possibly shared with other referrers**	may also refer to “no object” (like a `nullptr`)
Data Size	same as plain pointer	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



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*

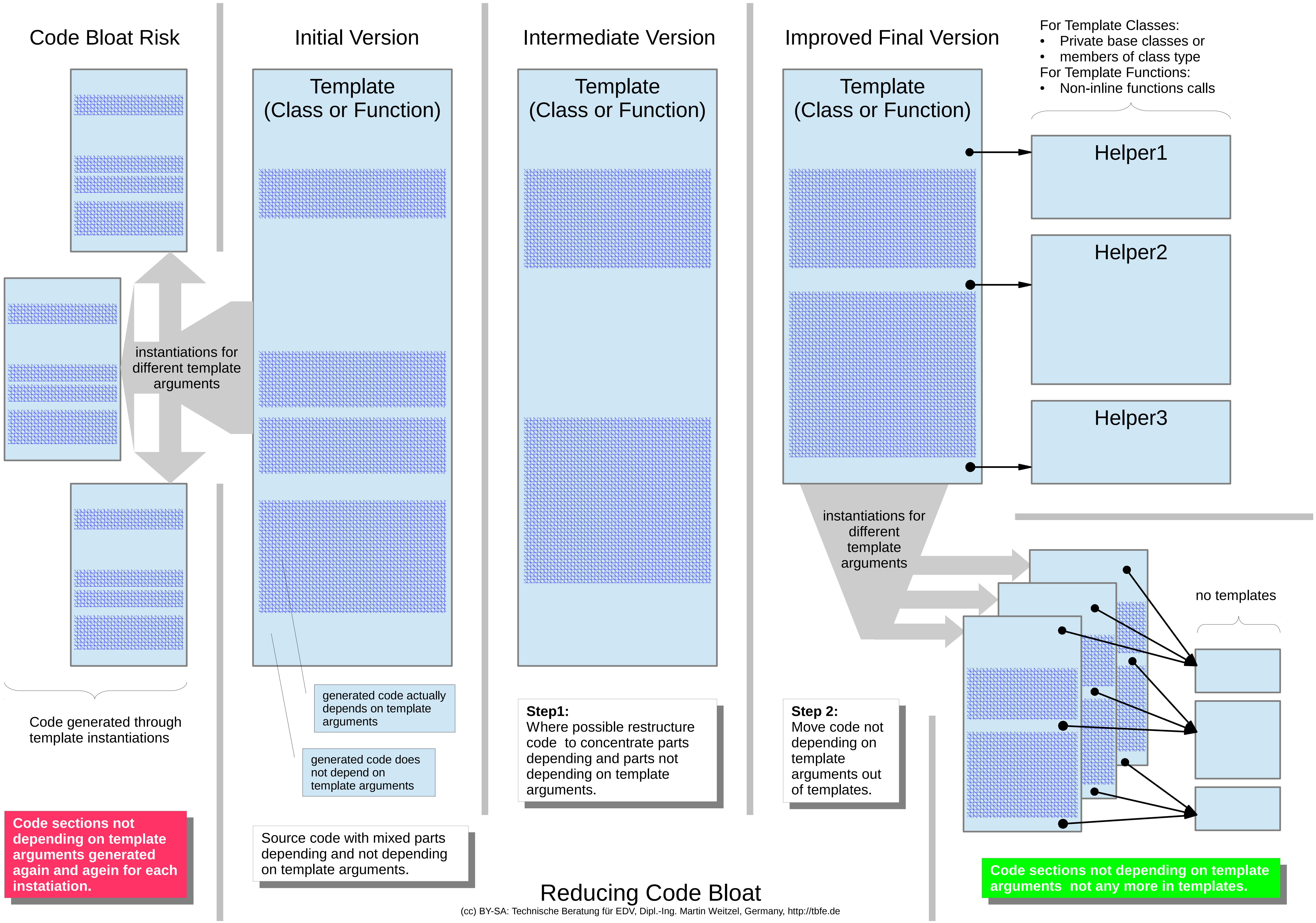
```
template<std::size_t Size>
class RingBuffer {
        double data[Size+1];
        …
        static std::size_t wrap(std::size_t idx) {
                return Size+1;
        }
        …
        std::size_t size() const {
                return (iput >= iget)
                        ? iput - iget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



Reducing Code Bloat