Effective C++ Programming

NIE-EPC (v. 2021):

EXCEPTION SAFETY, ALLOCATORS, SMALL BUFFER OPTIMIZATION, EMPTY BASE OPTIMIZATION, TYPE ERASURE

Vector — reallocation — *safety concerns*

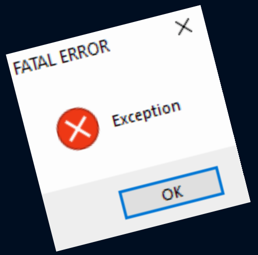
- *Not-exactly-final* solution for `reserve` member function:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));

  for (size_t i = 0; i < size_; i++)
    new (data + i) T( std::move( *(data_ + i) ) );  // what if throws ???

  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- In case of *non-trivially-copyable types*, initialization involves calls of copy or move constructor, which, generally, may throw exceptions.
- Such situations need to be handled:
  - some elements may have been already constructed before,
  - new storage has been already allocated.



- *Note* — for sake of simplicity, we ignored this issue previously.
- *Note* — initialization of objects of *trivially copyable types* with content-copying here cannot result in exception.

Vector — reallocation — *safety* (cont.)

- When initialization of *ith* element throws an exception, we need to "rollback" all the already performed operations:
  - Rollback already constructed elements = *their destruction*.
  - Rollback allocated memory = *deallocation*.

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));
  size_t i = 0;
  try {
    for ( ; i < size_; i++) new (data + i) T( std::move( *(data_ + i) ) );
  } catch (...) {
    for ( ; i > 0; i--)
      (data + i - 1)->~T();    // destruct already constructed new elements
    ::operator delete(data);   // deallocate storage for new elements
    throw;                     // rethrow caught exception to function caller
  }
  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- *Question* — will the state/content of the vector itself change?
- *Answer* — it depends.

Vector — reallocation — *safety* (cont.)

- If new elements were initialized while *copying content* from original elements, the answer is **NO**:
  - The original elements — which make vector content (state) — will remain preserved (copying content does not modify copied-from objects).
- *Problem* — if new elements were initialized while *moving content* form original elements, the answer is **YES**:
  - The original elements generally no longer have their original content — it was *moved-from them*.
  - Instead, they are in a "*moved-from*"/"*empty*" state.
- Is there any solution? No.
  - We may try to move content back from new elements (before their destruction) to the original ones.
  - However, since *moving-content* operation may throw, it may throw again during this "rollback".
  - ⟹ There is no guarantee that such a rollback will succeed.

Vector — reallocation — *safety* (cont.)

- *Conclusion:*
  - *Move constructors* and *move assignment operators* should better generally not throw exceptions.
  - If they do, users of such classes need to be aware of consequences.
  - *Example* — when move constructor throws during `Vector` reallocation, (some of) its elements may end up in a modified (*empty*/*moved-from*) state.
- *Alternative approach* — to create vectors more "*safe*" with respect to exceptions, we want, during reallocation:
  1) *move constructor* to be used for initialization of new elements if it is available and may not throw (is so-called "*non-throwing*");
  2) otherwise, *copy constructor* to be used if it is available;
  3) otherwise — when there is no other possibility — "*throwing*" *move constructor* to be used (*as a last unsafe instance*).

Vector — reallocation — *safety* (cont.)

- Solution in our `Vector<T>::reserve` code is to change…

```
for ( ; i < size_; i++) new (data + i) T( std::move( *(data_ + i) ) );
```

- …to:

```
for ( ; i < size_; i++) new (data + i) T( std::move_if_noexcept( *(data_ + i) ) );
```

- *Explanation:*
  - `std::move_if_noexcept` "changes a value category" of its argument to *rvalue* (which causes move constructor to be considered first) conditionally — namely, only if move constructor of `T` if *non-throwing*.
  - On the contrary, `std::move` changes a value category or its argument to rvalue *unconditionally* (in all cases).
- How to indicate a non-throwing constructor?

```
struct X {
  X(X&&) noexcept { ... }
};
```

- *Note* — if this "*non-throwing*" *specification* is violated (exception is thrown) program is exited via `std::terminate` call.

`std::vector` — reallocation — *safety*

- This "safer" approach is adopted by `std::vector`.
- *Illustrative benchmark:*
  - Insertion ("*push-backing*") of elements into a vector without pre-allocation of storage (`reserve`).
  - *Value types* — three variants of `String` class without applied SSO:
    - *Variant I.* — only copy constructor (*CC*).
    - *Variant II.* — copy constructor + throwing move constructor (*MC*).
    - *Variant III.* — copy constructor + non-throwing move constructor.
  - *Results* — comparison with *Variant I.* case:
    - *Variant II.* — **1.5× faster**.
    - *Variant III.* — **2.6× faster**.
  - *Link* — https://quick-bench.com/q/xv8LbXAwIhaFrR6IoQlwwHRuhEQ.
  - *Explanation*:
    - *Variant I.* — *CC* is used for both *insertion* and *reallocation*.
    - *Variant II.* — *MC* is used for *insertion*, *CC* is used for *reallocation*.
    - *Variant III.* — *MC* is used for both *insertion* and *reallocation*.

`std::vector` — realloc. — *safety (cont.)*

- Formulation from the C++ standard regarding `std::vector::reserve` (*reallocation*) — [link]:
  - *"If an exception is thrown other than by the move constructor of a non-Cpp17CopyInsertable type, there are no effects."*
- Translation to understandable form:
  - ⇒ *"Only if an exception is thrown ~~other than~~ by the move constructor of a non-Cpp17CopyInsertable type, there may be some effects."*
  - ⇒ *"Only if an exception is thrown by the move constructor of a non-copyable type, there may be some effects."*
  - ⇒ *"Only if an exception is thrown by the move constructor of a non-copyable type, the vector state/content may change."*
- *Consequence:*
  - ⇒ The only case when the vector state/content may change during reallocation is when its value type is non-copyable (has no copy constructor) and has throwing move constructor.
  - ⇒ Otherwise — if value types has either copy constructor or non-throwing move constructor, `reserve` that ends up with exception is guaranteed not to change the vector state.
- The same holds for inserting element at the end (`push_back`, `emplace_back`), when reallocation may happen — [link].

`std::vector` — realloc. — *safety (cont.)*

- Seemingly *surprising consequences*:

```
struct X {                   // non-copyable type
    X(const X&) = delete;
};

int main() {
  using T = std::map<int, X>;
  std::vector<T> v;
  v.reserve(100);
}
```

- This code:
  - compiles well with *libstdc++* and *libc++*,
  - results in compilation error with *Microsoft STL*.
  - *Live demo* — https://godbolt.org/z/j4h14Gfae.
- *Explanation:*
  - C++ standards do not prescribe move constructor of `std::map` to be *non-throwing* (noexcept).
  - However, implementations are allowed to strengthen exception specifications of library (member) functions.
  - *libstdc++* and *libc++* do make move constructor of `std::map` *non-throwing*.
  - *Microsoft STL* does not.

`std::vector` — realloc. — *safety (cont.)*

- *Recall* — initialization of new elements in `Vector::reserve`:

```
for ( ; i < size_; i++) new (data + i) T( std::move_if_noexcept( *(data_ + i) ) );
```

- Implementations of `std::vector::reserve` basically use effectively the same solution (*see later*).
  - ⇒ With *libstdc++* and *libc++*, the initialization expression inside `reserve` is resolved as (*non-throwing*) **move constructor** call.
  - ⇒ However, with *Microsoft STL* — due to throwing map (vector value type) move constructor — the initialization expression is resolved as **copy constructor** call.
- Copy constructor of `std::map` exists and tries to copy its content.
- This is not possible, since content of map include objects of type `X`, which is *non-copyable*.
- Namely, copy constructor of map tries to call copy constructor of `X`, which is *deleted*.
- *Note* — this is more generic problem:
  - Different compilation behavior with different implementations.
  - *Discussion*, for example — https://stackoverflow.com/q/65140603/580083.

`std::vector` — realloc. — *safety (cont.)*

- *Error message* of MSVC — 126 lines, 10909 characters.
- Important part:

```
error C2280: 'std::pair<const int,X>::pair(const std::pair<const int,X> &)': attempting to
reference a deleted function

note: see declaration of 'std::pair<const int,X>::pair’

note: 'std::pair<const int,X>::pair(const std::pair<const int,X> &)': function was
implicitly deleted because a data member invokes a deleted or inaccessible function
'X::X(const X &)’

note: 'X::X(const X &)': function was explicitly deleted
```

- It is very hard to find out the cause of the problem here.
  - Which is potentially throwing move constructor of `std::map`.
- Different behavior with different implementations kind of indicates that this is some bug in Microsoft STL.
  - Actually, it is not.
- Could *concepts/constraints* help here?
  - In theory yes, but even then it would not be straightforward.
  - At best, copy constructor of map could be “disabled” for non-copyable mapped type (X in our case).

Exception safety guarantees

- Important specification regarding some operation (such as function call) with respect to exceptions is:
  - whether they may throw exceptions,
  - and if they do, what may happen with the program state (typically, state of the involved objects).
- *Possibilities:*
  - *No-throw guarantee* — exceptions may not be thrown.
  - *Strong exception guarantee* — in case of exception, the program state will not change (operation has *no effect*).
  - *Basic exception guarantee* — in case of exception, the program state may change but remains valid/correct.
  - *No exception guarantee* — in case of exception, anything may happen (no guarantee about the program state $\Rightarrow$ basically, it cannot continue to run safely).

Exception safety guarantees — *examples*

- *Example:*
  - With the *new terminology*, `std::vector` member functions `reserve`, `push_back`, and `emplace_back` provide strong exception guarantee if its *value type* has either copy constructor or non-throwing move constructor.
- *Example:*
  - `std::vector::clear` for *non-trivially-copyable* types involve destructor calls.
  - These destructors may, generally, throw exceptions (don't write such ones).
  - However, `clear` has noexcept specification ⇒ it provides no-throw guarantee.
  - ⇒ If exceptions are thrown by destructors, they must be ignored and not propagated outside of `clear` function call (which is very unsafe).

- *Example:*
  - Destructors of all library types provide no-throw guarantee:
    - "Destructor operations defined in the C++ standard library shall not throw exceptions." [link]
  - *Implication* — for owner types, exceptions from destructors of owning objects are ignored.

std::vector — reallocation — *allocators*

- Final version — *really*?

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));

  size_t i = 0;
  try {
    for ( ; i < size_; i++)
      new (data + i) T( std::move_if_noexcept( *(data_ + i) ) );  // can be replaced by...
        // ... std::contruct_at<T>(data + i, std::move_if_noexcept(*(data_ + i))); ...
        // ... since C++20
  } catch (...) {
    for ( ; i > 0; i--)
      (data + i - 1)->~T();  // can be replaced by std::destroy_at(data + i – 1) since C++17
    ::operator delete(data);
    throw;
  }

  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- Does `std::vector` works this way?
  - *By default* — **YES**.
  - *Generally* — **NO**.

`std::vector` — realloc. — *allocators (cont.)*

- `std::vector` is a so-called "*allocator-aware*" container:
  - *Dynamic storage allocation/deallocation* are not performed directly by C++ allocation/deallocation functions (`operator new` and `operator delete`).
  - *Object initialization* is not performed directly by *placement-new construct*.
  - *Object destruction* is not performed directly by *(pseudo-)destructor call*.
- *Instead:*
  - An ***allocator type*** is provided as a second `std::vector` template argument.
  - This allocator type is then used to resolve how:
    1) storage will be allocated/deallocated,
    2) elements will be initialized/destructed.

std::vector

Defined in header `<vector>`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

- *Default template argument* for allocator type template parameter of `std::vector<T>` is `std::allocator<T>`.

`std::vector`—realloc.—`std::allocator`

- With `std::allocator<T>`:
  - *Storage is allocated* by `std::allocator<T>::allocate` member function, which allocates memory by `::operator new`.
    - *Link* — https://en.cppreference.com/w/cpp/memory/allocator/allocate.
  - *Storage is deallocated* with `std::allocator<T>::deallocate` member function, which deallocates storage with `::operator delete`.
    - *Link* — https://en.cppreference.com/w/cpp/memory/allocator/deallocate.
  - *Elements are initialized* with `std::allocator_traits<std::allocator<T>>::construct`, which initializes object with *placement new* (or, `std::construct_at` since C++20).
    - *Link* — https://en.cppreference.com/w/cpp/memory/allocator_traits/construct.
  - *Objects are destructed* with `std::allocator_traits<std::allocator<T>>::destroy`, which destructs them with *(pseudo-)destructor call* (or, `std::destroy_at` since C++20).
    - *Link* — https://en.cppreference.com/w/cpp/memory/allocator_traits/destroy.
- ⇒ *By default* — that is with `std::allocator<T>` — `std::vector` works during reallocation effectively the same as our `Vector`.

`std::vector`—realloc.— *custom allocators*

- However, *customization* of all these *“special” allocator operations* regarding storage and object-lifetime management is possible by providing custom allocator types.
- *Illustrative use case I.:*
  - *SIMD processing* — stronger than default alignment is required (such as 64-byte for AVX-512).
  - ⇒ Custom `allocate` function can provide this.
  - Instead of “ordinary” `operator new`, it may use:
    - *alignment-aware heap functions* (such as `posix_memalign`, `aligned_alloc`,…),
    - special versions of `operator new` that support alignment requests (available since C++17).
  - *Relevant discussion:* [Making std::vector allocate aligned memory]

`std::vector` — *custom allocators (cont.)*

- *Illustrative use case II.:*
  - For *trivially copyable* types, standard allocator causes "*zeroing out*" binary representations of vector elements when vector is "*resized*".
  - For large vectors, it may take some time (*runtime overhead*).
  - This is unnecessary if elements will then be rewritten anyway.
    - A use case is, for example, filling vector content from multiple threads (element-insertion member functions cannot be called from multiple threads at once ⇒ vector needs to be "*resized*" in advance).
  - Custom allocator with *do-nothing* `construct` function may resolve this issue.
  - *Relevant discussions:*
    - [Default-inserting into a vector isn't default initialization?]
    - [Should allocator construct() default initialize instead of value initializing?]

Other containers — *allocators*

- *Illustrative use case III.:*
  - `std::vector` is not the only one allocator-aware data structure from the C++ library.
  - Node-based allocator aware containers (*sets*, *maps*, their *unordered variants*) may benefit from allocators that use ***memory pooling***.
  - *Memory pooling* employs the fact that each node requires storage of the same size.
    - Memory pool-based allocators "*merge*" heap allocations for storage of multiple nodes into single allocation.
    - ⇒ Customization of `allocate` allocator function.
  - Moreover, if a container is *thread-local*, memory-pool allocators may be *thread-local* as well (no need for synchronization between threads).
    - *Benchmark* — **1.3× faster** insertion into *thread-local* hash tables (see NI-MCC for more details.)
  - *Real-world implementations:*
    - `pool_allocator` or `fast_pool_allocator` provided by *Boost.Pool library* [link].

Vectors and SBO

- *Recall* — *short string optimization* (SSO) = up to some limit length (number of characters), strings are stored
  - in *included storage* of the string class owner (its member variable buffer)
  - instead of in *dynamically-allocated storage*.
- *String class* = owner of dynamic array of *characters*.
- *Vector class* = owner of dynamic array of *elements of any type* (value type).
- Could vectors employ the same optimization technique?
  - Generally, it is called *small buffer (data/object/size) optimization* (SBO).
- SBO for vectors — we would want:
  - $\Rightarrow$ Vectors to have included storage (member variable buffer) with *capacity* of some small number of vector elements.
  - Store vector into this buffer until their count (vector size) does not exceed this "*small buffer*" capacity.

Vectors and SBO *(cont.)*

- Generally, such optimization technique can be implemented.
- However — unfortunately — not for `std::vector`.
- There are requirements in the C++ standard that hinder application of SBO for `std::vector`:
  - *Exemplary problem case* — swapping content of two vectors.
  - *Corresponding requirement* — "*no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped*" [link].
  - *Problem* — with SBO, for instance, swapping elements of "*short*" and "*long*" vectors do invalidate references, pointers, and iterators to elements in the included storage of the short vector.
- In theory, removing the above-quoted requirement from upcoming C++ standards might enable SBO for `std::vector` in the future.
- However, it could make the existing C++ code incorrect, which is not acceptable.
  - New C++ standards break backward compatibility in extremely rare cases.

Vectors and SBO *(cont.)*

- There are implementations of *vector-like classes* that do implement SBO.
- ⇒ They are not 100% compatible with `std::vector` and their users need to be aware of it.
- *Exemplary implementation:*
  - `small_vector` class template from *Boost.Conatiner library* [link].
  - In comparison with `std::vector`, it has an additional template non-type parameter — *capacity of the included storage* (buffer).
  - ⇒ Users may control *short* vectors limit sizes according to their needs.
- Another commonly-used alternative:
  - *LLVM's* `SmallVector` [link].
- *Target use cases* for vectors with SBO:
  - Programs that work with many vectors with unknown sizes (until runtime), where many of them has **only few elements**.

Empty base (class) optimization

- Assume we wanted to make our custom `Vector` *allocator-aware*.
- First consideration — allocator objects may have state.
  - ⇒ The same allocator needs to be used through entire vector lifetime.
- Possible implementation:

```
template <typename T, typename A = std::allocator<T> > class Vector {
  size_t capacity_, size_;
  T* data_;
  A alloc_;  // allocator member sub-object

public:
  Vector( const A& alloc = A{} )
    : capacity_(0), size_(0), data_(nullptr), alloc_(alloc) { }
```

- *Problem* — even when allocator is state-less (it has no sub-objects), it causes increase of `Vector` storage size:

```
std::cout << sizeof( Vector<int> );          // prints out 32 (originally, it was 24)
```

- Two causes:
  1) Generally, different objects cannot share storage ⇒ `alloc_` member variable occupies at least 1 byte.
  2) Due to *alignment* and *padding*, at least 8 bytes are effectively needed.

Empty base (class) optimization

- Ad 1) Consider the following example:

```
X a[10];  // array of 10 elements (objects) of type X
```

- According to the C++ standard, *size of storage/binary representation of an object of type* X — that is `sizeof(X)` — equals (byte) difference between addresses of two adjacent array elements.
- These two elements cannot share (be stored at) the same address.
- ⇒ They need to be stored at different addresses.
- ⇒ `sizeof(X)` must be at least 1.
- Generally, this holds for any type, even when it is "*empty*" (has no member/base-class sub-objects):

```
struct X { };
```

```
std::cout << sizeof(X);  // prints out 1
```

- This is also the case of the standard library allocator, which is *state-less* ⇒ does not require any sub-objects:

```
std::cout << sizeof( std::allocator<int> );  // prints out 1
```

Empty base (class) optimization

- ⇒ When allocator is a member sub-object/variable of `Vector`, even if it is state-less, it occupies at least 1 byte.
- Due to alignment and padding, it effectively occupies 8 bytes.
- Another option — *inheritance*:

```
template <typename T, typename A = std::allocator<T> > class Vector : public A {
  size_t capacity_, size_;
  T* data_;

public:
  Vector( const A& alloc = A{} )
    : capacity_(0), size_(0), data_(nullptr), A(alloc) { }
```

```
std::cout << sizeof( Vector<int> );          // now, prints out 24 (GCC, x86_64)
```

- How is that possible?
  - It is caused by an optimization mechanism called "*empty base (class) optimization*" (EBO/EBCO).
  - *Meaning* — inheritance from an "empty" class does not increase storage/binary representation requirements of the derived class.
  - This optimization is not mandatory, but all mainstream implementations do apply it.

Public vs private inheritance

```
template <typename T, typename A = std::allocator<T> > class Vector : public A {
```

- *Note* — *public inheritance* represents the IS-A relationship.
- In our case, this is wrong — vector is NOT an allocator.
- *Unwanted consequence* — public member functions of allocator type are "injected" into the `Vector` interface.
  - Such as `allocate` member function might be called on a `Vector` object.
- Better version — *private inheritance*:

```
template <typename T, typename A = std::allocator<T> > class Vector : private A {
```

- *Notes:*
  - Inheriting `Vector` from allocator type kind of "*smells*" and can make its code less understandable.
  - We implemented it only because of EBO.
  - For this reason, since C++20, we can have the same outcome even with *composition* and no_unique_address attribute:

```
template <typename T, typename A = std::allocator<T> > class Vector {
  size_t capacity_, size_;
  T* data_;
  [[no_unique_address]] A alloc_;  // better than inheritance, but since C++20
```

Empty base (class) optimization

- Another use case — *unique pointers*:
  - Standard library unique pointers support so-called custom *deleters*.
  - *Deleter* is an object that defines function-call operator, which is applied to the managed pointer when unique pointer itself is destroyed.
- Type of *deleter* makes second template argument of `std::unique_ptr`.
- *By default*, it is `std::default_delete`, which just applies `delete` *expression* to the managed pointer.

std::unique_ptr

Defined in header `<memory>`

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

- Illustrative custom implementation:

```
template <typename T> struct default_delete {  void operator()(T* ptr) { delete ptr; }  };

template <typename T, typename D = default_delete<T>> class unique_ptr {
  T* ptr_;
  D del_;

public:
  unique_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), del_(del) { }
  ~unique_ptr() { del_(ptr_); }
};
```

Empty base (class) optimization

```
template <typename T> struct default_delete {  void operator()(T* ptr) { delete ptr; }  };

template <typename T, typename D = default_delete<T>> class unique_ptr {
  T* ptr_;
  D del_;
```

- Again — with this implementation, *storage size* of unique pointer objects with *default deleter* would grow from 8 to 16 bytes (at 64-bit architecture)…
- … even if this *default deleter* is *state-less* (has no sub-objects).
- This is highly unwanted — unique pointers should be just efficient wrappers about normal (raw) pointers.
- *Solution* (before C++20):

```
template <typename T, typename D = default_delete<T>> class unique_ptr : private D {
  T* ptr_;

public:
  unique_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), D(del) { }
  ~unique_ptr() { D::operator()(ptr); }
};
```

Custom deleters

- Default deleter works for object dynamically-allocated and initialized by the new *expression*:

```
{
  std::unique_ptr<X> upx = new X{};

} // delete expression applied automatically to the owned pointer to the allocated object
  // ...in the destructor of upx
```

- Custom deleter allows applying different operation than `delete` *expression*.
- *Example* — MPI I/O (see NI-PDP for more details):
  - Part of MPI library that provide parallel access to a single file from multiple MPI processes.
- MPI — C API:
  - MPI file handler has type `MPI_File`…
  - …and its address is passed to `MPI_File_open` and `MPI_File_close` functions.

```
MPI_File f;
MPI_File_open(..., &f);

...  // read/write from/to file simultaneously by multiple MPI processes

MPI_File_close(&f);
```

Custom deleters *(cont.)*

- *Problem* — in case of exception thrown before `MPI_File_close`, the MPI file handle will not be closed, which may result in loos of data.

```
MPI_File f;
MPI_File_open(..., &f);
... // What if throws?
MPI_File_close(&f);
```

- Solution = *RAII idiom*:
  - *RAII* = *resource acquisition is initialization*.
  - *Meaning* — responsibility for resource release is put into a destructor of some block-local object.
  - Its destructor is called automatically whenever program leaves that block (reaching its end, `return` statement, exception, `goto`,…).
- *Unique pointers* — by default — apply RAII for objects dynamically allocated and initialized by new expression.
- By defining custom deleter, we can use them for guaranteed correct MPI file handle closing:

```
struct MPI_deleter {
  void operator()(MPI_File* ptr) {
    MPI_File_close(ptr);
  }
};
```

```
MPI_File f;
MPI_File_open(..., &f);
std::unique_ptr<MPI_File, MPI_deleter> temp = &f;
...
// anytime temp is destructed, file is closed
```

Custom deleters — shared pointers

- Same as with *unique pointers*, *shared pointers* also support *custom deleters*.
- In contrast to unique pointers, type of deleter does not make a template argument of the shared pointer:

```
template <typename T, typename D = default_delete<T>>
class unique_ptr { D del_; ... };      // option #1 - composition (member sub-object)

template <typename T, typename D = default_delete<T>>
class unique_ptr : private D { ... };  // option #2 - inheritance (base-class sub-object)

template <typename T> class shared_ptr { ... };  // ???
```

- *Questions/problems:*
  1) How is a deleter set for shared pointers?
  2) How to store it if its type is unknown at a class scope?
- Ad 1) Similarly as for unique pointers, a deleter object is set in constructor.
  - ⇒ For shared pointers, constructor must be parametrized by deleter type:

```
template <typename D = std::default_delete<T>>
shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr) ...   // ???
```

Custom deleters — shared pointers *(cont.)*

```
template <typename T> class shared_ptr {
  T* ptr_;
  ...  // reference-counting housekeeping (pointer to control block); omitted later
public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr) ...   // ???
```

- ⇒ Deleter object and its type are known in the constructor only.
- Its copy need to be stored somewhere and managed by the shared pointer.
- It cannot be stored as a sub-object (in the included storage).
- But, it can be dynamically allocated:

```
template <typename T> class shared_ptr {
  T* ptr_;
  ???* ptr_del_;
public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), ptr_del_(new D{del}) { }
```

- *Problem* — what type should that pointer be of, when we don't know the deleter type at the class scope where it is declared?

Custom deleters — shared pointers *(cont.)*

- *Seemingly possible solution:*
  - "Type-less" pointer, that is pointer of type `void*`.
  - Any object may be pointed to by a pointer of type void*.

```
template <typename T> class shared_ptr {
  T* ptr_;
  void* ptr_del_;

public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), ptr_del_(new D{del}) { }

  ~shared_ptr() { if (use_count() == 1) { ??? } }

  long use_count() const
  {  ... /* returns number of shared_ptr instances managing current object */  }
```

- *Problem* — in destructor of shared_ptr, we possibly need to:
  - *apply the deleter* to the ptr_ pointer,
  - *apply* `delete` *expression* to the deleter itself.
- Neither can be done without knowing the deleter type.
  - We need to cast `ptr_del_` to `D*` in destructor, where `D` is unknown.
- ⇒ This is *dead end*.

Custom deleters — shared pointers *(cont.)*

- Another option — storing a deleter in a "wrapper" helper class:

```
template <typename T> class shared_ptr {
  template <typename D> struct Helper
  {  D del_;
     Helper(const D& del) : del_(del) { }  };
  T* ptr_; ???* ptr_del_;  // D unknown here
public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), ptr_del_(new Helper<D>{del}) { }
```

- *Problem* — we cannot make `ptr_del_` to have `Helper<D>*` type.
- However, there is a case where pointer of some type can point to an object of another type — *inheritance*.
  - Namely, *pointer-to-base* class can point to a *derived class object*.
- In our case, this means to:
  - create a common base class,
  - inherit `Helper` from this base class,
  - make `ptr_del_` a pointer to base.

```
template <typename T> class shared_ptr {
  struct Base { };
  template <typename D>
  struct Helper : Base {
    D del_;
    Helper(const D& del) : del_(del) {}
  };
  T* ptr_;
  Base* ptr_del_;
```

Custom deleters — shared pointers *(cont.)*

```
template <typename T> class shared_ptr {
  struct Base { };
  template <typename D>
  struct Helper : Base {  D del_; Helper(const D& del) : del_(del) {}  };
  T* ptr_; Base* ptr_del_;
public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), ptr_del_(new Helper<D>{del}) { }
  ~shared_ptr() { if (use_count() == 1) {
     // (1) apply operator() of deleter to ptr_
     // (2) delete created Helper<D> class object
   } }
```

- What remains to be done:
  - Object of type `Helper<D>` created in constructor needs to be “deleted”.
  - ⇒ Application of `delete` expression to `ptr_del_` must invoke destructor of the `Helper<D>` class.
  - ⇒ This destructor needs to be virtual.
  - *Note* — it also destructs deleter itself.

```
template <typename T>
class shared_ptr {
  struct Base {
    virtual ~Base() = default;
  };
  ...
  ~shared_ptr() {
    if (use_count() == 1) {
     // (1)
     delete ptr_del_;
    }
  }
```

Custom deleters — shared pointers *(cont.)*

- *Last problem* — application of deleter stored in the derived class object to ptr_ in destructor of shared_ptr.
  - Here, only pointer to base is available.
  - ⇒ We again need to involve a virtual function mechanism.

```
template <typename T> class shared_ptr {

  struct Base {
    virtual ~Base() = default;
    virtual void operator()(T* ptr) = 0;
  };

  template <typename D>
  struct Helper : Base {
    D del_;
    Helper(const D& del) : del_(del) {}
    virtual void operator()(T* ptr) override { del_(ptr); }
  };

  T* ptr_; Base* ptr_del_;

public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{}) : ptr_(ptr), ptr_del_(new Helper<D>{del}) { }

  ~shared_ptr() { if (use_count() == 1) {
     ptr_del_->operator()(ptr_);  // same as (*ptr_del_)(ptr_);
     delete ptr_del_;
   } }
```

- *Note* — Base is not meant to be instantiated ⇒ it is made *abstract*.

Custom deleters — shared pointers *(cont.)*

- *Exemplary instantiation* with value type X, default deleter, and its custom `default_delete` implementation:

```
shared_ptr<X> ptr = new X;
```

```
struct default_delete<X> {  void operator()(X* ptr) { delete ptr; }  };

class shared_ptr<X> {

  struct Base {
    virtual ~Base() = default;
    virtual void operator()(X* ptr) = 0;
  };

  struct Helper<default_delete<X>> : Base {
    default_delete<X> del_;
    Helper(const default_delete<X>& del) : del_(del) {}
    virtual void operator()(X* ptr) override { del_(ptr); }
  };

  X* ptr_;
  Base* ptr_del_;

public:
  shared_ptr<default_delete<X>>(X* ptr, const default_delete<X>& del = default_delete<X>{})
    : ptr_(ptr), ptr_del_(new Helper<default_delete<X>>{del}) { }

  ~shared_ptr() { if (use_count() == 1) {
     ptr_del_->operator()(ptr_);
     delete ptr_del_;
   } }
```

Custom deleters — shared pointers *(cont.)*

- In case of additional instantiation with the same value type X, but a different custom deleter, the resulting instance will look like:

```
shared_ptr<X> ptr = new X;   shared_ptr<X> ptr2 { new X, custom_delete<X>{} };
```

```
struct default_delete<X> {  void operator()(X* ptr) { delete ptr; }  };
struct custom_delete<X> {  void operator()(X* ptr) { ... }  };

class shared_ptr<X> {

  struct Base {
    virtual ~Base() = default;
    virtual void operator()(X* ptr) = 0;
  };

  struct Helper<default_delete<X>> : Base {
    default_delete<X> del_;
    Helper(const default_delete<X>& del) : del_(del) {}
    virtual void operator()(X* ptr) override { del_(ptr); }
  };

  struct Helper<custom_delete<X>> : Base {
    custom_delete<X> del_;
    Helper(const custom_delete<X>& del) : del_(del) {}
    virtual void operator()(X* ptr) override { del_(ptr); }
  };

  X* ptr_;
  Base* ptr_del_;

public:
  shared_ptr<default_delete<X>>(X* ptr, const default_delete<X>& del = default_delete<X>{})
    : ptr_(ptr), ptr_del_(new Helper<default_delete<X>>{del}) { }

  shared_ptr<custom_delete<X>>(X* ptr, const custom_delete<X>& del = custom_delete<X>{})
    : ptr_(ptr), ptr_del_(new Helper<custom_delete<X>>{del}) { }

 ~shared_ptr() { if (use_count() == 1) { ptr_del_->operator()(ptr_); delete ptr_del_; } }
```

Deleters — unique vs shared ptrs.

- *Observation:*
  - Type of all `shared_ptr` objects with the same value type but different deleter types **is the same**.
    - It is the same class = instance of `shared_ptr` class template.
  - Different type of deleter just causes different instance of shared pointer constructor to be generated within this class (and, possibly, some implementation inner helper class).
  - On the contrary, for each type of deleter `unique_ptr` class template is **instantiated into different class**.
    - Types of unique pointers with different deleters are different and unrelated classes.
- In shared pointers, type of deleter is so-called "*type-erased*".
  - The technique for hiding its type is called "*type erasure*".
- Type-erased deleter is more "powerful" but has runtime overhead.
  - $\Rightarrow$ Reason why unique pointers do not employ this technique.
    - *Recall* — unique pointer = lightweight RAII pointer wrapper with minimal overhead.