Effective C++ Programming

NIE-EPC (v. 2021):

COSTS OF DYNAMIC ALLOCATIONS, ALIGNMENT, PADDING, PLACEMENT NEW, OWNING OBJECTS

Runtime costs of dynamic allocations

- Statically and dynamically allocated objects differ only in the way how their storage is (de)allocated.
  - Initialization and destruction is the same.
- Dynamic allocations typically involve a *heap* — a very complex mechanism that needs to be able to:
  - satisfy request for allocations of different sizes,
  - communicate with a kernel to ask for memory blocks (such as pages),
  - synchronize request in multi-threaded programs (which, generally, brings some overhead into single-threaded programs as well),
  - …
- $\Rightarrow$ Allocation of the storage on the heap is a relatively complex task, when compared, for instance to allocation on the stack.
  - *Recall* — allocation on the stack involves just decrementing the stack pointer register (which is "super-fast").

Runtime costs of dynamic allocs (*cont.*)

- *Experiment* — comparison of static and dynamic allocation of a storage for an object of type `int`:

```
// I. static allocation experiment:
int i = 1;
...  // make sure storage for i is allocated in memory (on the stack)
```

```
// II. dynamic allocation experiment:
int* pi = new int(1);
...  // make sure storage for *pi is allocated in memory (on the heap)
```

- *Benchmark:*
  - Used tool — *Google Benchmark*, namely its online form *Quick C++ Benchmark*.
  - Repeated execution in the loop.
  - Relative comparison of the average time of a single iteration.
  - *Link:* https://quick-bench.com/q/d_ENZrzvO_jwby2Qy7bisugNVr8.
- *Results:*

  - *Dynamic allocation* was **20-40× slower** than *static allocation*.

Memory costs of dynamic allocations

- Is slower performance the only drawback of dynamic storage allocation?
- Unfortunately, NO :(
- Another disadvantage — memory overhead.
- Two causes:

1) ***Alignment:***
   - Padding caused by alignment produces wasted memory bytes.
2) ***Housekeeping data:***
   - Heap needs some auxiliary data to be stored in memory for each allocation to keep track of them.

- *Effect* — with typical heap implementations:
  - Each allocation — even of a single byte — **consumes at least 32 bytes** of memory on 64-bit systems.

Storage alignment

- Storage for an object (= each object) of type `T` needs to be aligned to `alignof(T)`-byte address.
  - Alignment to an N-byte address means alignment to an address A, where $A = 0$ (modulo N).
- Alignment requirements for particular types are implementation-defined.
- *Example — GCC/Linux/x86_64*:

```
std::cout << alignof( char        );  // prints out  "1"
std::cout << alignof( int         );  // prints out  "4"
std::cout << alignof( long        );  // prints out  "8"
std::cout << alignof( float       );  // prints out  "4"
std::cout << alignof( double      );  // prints out  "8"
std::cout << alignof( long double );  // prints out "16"
std::cout << alignof( void*       );  // prints out  "8"
```

- *Class types* — alignment requirements typically equal the maximum of alignment requirements for types of its subobjects.
- *Note: Subobjects = non-static member variables* and *base class objects*.

Storage alignment (*cont.*)

- **Fundamental alignment** (*simplified definition*):
  - Alignment less than or equal to the maximum alignment for "fundamental" language-intrinsic types.
  - Is equal to the `alignof(std::max_align_t)` expression.
  - In typical C++ implementations, it equals alignment requirements for the `long double` type:

```
std::cout << alignof( long double      );  // prints out "16"
std::cout << alignof( std::max_align_t );  // prints out "16"
```

- **Over-aligned types**:
  - Types with alignment requirements that are higher than the maximal fundamental alignment.
  - *Exemplary use cases:*
    - *Cache line aligned data* (efficiency reasons, avoiding *false sharing*,…) — typically, 64-byte alignment.
    - *SIMD-processed data* (data processed by vectorization instructions), for example — AVX2 (32-byte alignment), AVX512 (64-byte alignment),…

Statically-allocated storage alignment

- How is alignment provided when a storage for an object is allocated?

1) Function non-static local variables (*automatic storage duration*) + typical stack-based implementation:
   - *Recall* — storge allocation = decreasing stack pointer (SP) register.
   - How much it needs to be decreased?
     - For example, decreasing by 4 bytes does not guarantee that SP will point to a 4-byte aligned address.
     - Generally, calculation of a properly aligned address would need additional instructions.
     - This would impose into functions additional (static) allocation overhead.

- Common real-world solution of this problem:
   - ABI prescribes that when the function is executed, SP must point to an A-byte aligned address, where A is defined in that ABI.
   - *Example* — *Linux/x86_64* — $A = 16$:
     - When the `call` instruction calls some function, RSP must be 16-byte aligned.

Statically-allocated storage alignment (*cont.*)

- *Example:*

```
void g(int*);

void f() {
  int i = 1;
  g(&i);
}
```

```
f():
  push   rax
  mov    dword ptr [rsp + 4], 1
  lea    rdi, [rsp + 4]
  call   g(int*)
  pop    rax
  ret
```

- When the function `f` is called somewhere, rsp is guaranteed to be 16-bytes aligned ⇒ rsp = 0 modulo 16.
- The corresponding `call` instruction decreases rsp by 8 (it "pushes" the return instruction address to the stack).
  - ⇒ When the function `f` is started, rsp = 8 modulo 16.
- Before g is called, rsp needs to be again 16-byte aligned.
  - ⇒ rsp needs to be decreased by another 8 bytes.
- Those 8 bytes are used for allocation of storage for `i`:
  - They are guaranteed to be 8-byte aligned, int requires 4-byte alignment.
  - ⇒ Storage for `i` can be allocated in upper 4 bytes or lower 4 bytes.
  - In our case, compiler decided to use the upper 4 bytes (rsp+4).
  - Lower 4 bytes will remain unused (wasted).

Statically-allocated storage alignment (*cont.*)

- *Modified example:*

```
void g(long*);

void f() {
  long i = 1;
  g(&i);
}
```

```
f():
  push  rax
  mov   qword ptr [rsp], 1
  mov   rdi, rsp
  call  g(long*)
  pop   rax
  ret
```

- Storage for an object of type `long` takes 8 bytes and requires 8-byte alignment ⇒ there are no bytes wasted.
- *Summary:*
  - Thanks to the ABI-required stack pointer alignment when functions are called, (static) allocations of storage for function-local variables can be performed extremely efficiently — namely, by decreasing stack pointer.
  - This holds for all *non-over-aligned* types.
  - For *over-aligned* types, some additional calculation at runtime is still needed (for instance, 16-byte alignment may or may not be 32-byte aligned at the same time).

Dynamically-allocated storage alignment

- How is alignment provided when a storage for an object is allocated (*cont.*)?

2) Dynamically-allocated objects (*dynamic storage duration*) + typical heap-based implementation:
   - *Recall* — storge allocation = calling some heap allocation function (through `operator new` C++ allocation function).
   - Typically, `malloc` is used.

- `malloc` takes a single argument — number of allocated bytes.
  - ⇒ It allows a caller to specify the storage (byte) size :).
  - ⇒ It does not allow a caller to specify alignment requirements :(.
- Common real-world solution of this problem:
  - Each dynamically-allocated block of memory is guaranteed to be aligned at least to the *maximal fundamental alignment*.

Dynamically-allocated storage alignment (cont.)

- Our *Linux/x86_64* system:
  - Heap implementation is (by default) provided by *GNU C library* (*GLIBC*).
    - This is a common case in Linux.
  - *Maximum fundamental alignment* `alignof(std::max_align_t)` is 16.
  - Accordingly, `malloc`-allocated memory blocks are guaranteed to be 16-byte aligned.
    - *Note* — heap implementations may guarantee even higher/stronger alignment (for details, see `__STDCPP_DEFAULT_NEW_ALIGNMENT__` C++ macro).
- What about over-aligned types?
  - Since C++17, `operator new` has overloaded versions where alignment requirements may be specified.
    - Internally, other function than `malloc` needs to be used by a C++ standard library implementation (such as the `posix_memalign` allocation function).
  - Before C++17, there was no portable way to dynamically-allocate storge for over-aligned types.

Memory costs of dynamic allocations (*cont.*)

- Consequences (*our platform*):
  - All dynamic allocations are 16-byte aligned.
  - ⇒ Even if a storage for an object of some type requires only 1 byte, its dynamic allocation will effectively consume 16 bytes.
  - The remaining bytes are wasted.
- Moreover, heap implementations require for each allocation some housekeeping data ⇒ additional memory overhead.
  - In our case, these data take another 16 bytes per allocation.
- *Experiment:*

```
void g(int*);  // NOOP function defined in another TU

int main() {
  for (int i = 0; i < 100'000'000; i++) {
    int* pi = new int{};
    g(pi);
} }
```

  - "Effective data" = 100M objects of type `int` ⇒ **400M bytes** of storage.
  - Measured memory consumption (*maximum RSS*): **3200M bytes**.
  - ⇒ Each allocation consumes 32 bytes:
    - 4 effective bytes (storage of `int` object), 12 bytes wasted, 16 bytes housekeeping.

Intermezzo — padding

- *Previous observation* — alignment can cause wasted bytes in memory in case of both:
  - static storage allocation (on the *stack*),
  - dynamic storage allocation (on the *heap*).
- These bytes are *out of* the allocated storage itself.
- Alignment requirements can cause wasted bytes even inside storage of a single object:
  - In the storage of a class type object, its *subobjects* are stored (*base class objects* and *non-static member variables*).
  - Due to alignment requirements, they may not be store next to each other in memory.
  - “Gaps” between subobjects are referred to as ***padding***.
  - Padding represents wasted bytes inside the storage of a single object.

Intermezzo — padding (*cont.*)

- *Example — Clang/x86_64/Linux:*

```
struct X { void* v; char c; int i; float f; };
```

- Storage for an object of type `X` starts at address A.
- `X::v` requires 8-byte aligned storage, which is 8 bytes long.
  - $\Rightarrow$ A is 8-byte aligned.
  - $\Rightarrow$ `X::v` is stored from address A to $A + 7$.
- `X::c` requires 1-byte aligned storage, which is 1 byte long.
  - $\Rightarrow$ `X::c` is stored at address $A + 8$ only.
- `X::i` requires 4-byte aligned storage, which is 4 byte long.
  - $\Rightarrow$ `X::i` cannot be stored next to `X::c` at address $A + 9$; this address is not 4-byte aligned.
  - $\Rightarrow$ The lowest 4-byte aligned address where `X::i` may be stored is $A + 12$.
  - $\Rightarrow$ `X::i` is stored from address $A + 12$ to $A + 15$.
  - Bytes from $A + 9$ to $A + 11$ represent padding (are *wasted*).
- `X::f` requires 4-byte aligned storage, which is 4 byte long.
  - $\Rightarrow$ `X::f` is stored from address $A + 16$ to $A + 19$.

Intermezzo — padding (*cont.*)

```
struct X { void* v; char c; int i; float f; };
```

- *Previous analysis:* Subobjects — member variables — `X::v`, `X::c`, `X::i`, and `X::f` are stored at addresses A to $A + 19$.
- If two objects of type `X` are stored in an array next to each other, and the first one is stored at address A, is the next element stored at $A + 20$? **NO!**
  - Its first member `X::v` needs to be 8-byte aligned.
  - $\Rightarrow$ The lowest available 8-byte aligned address higher than $A + 19$ is $A + 24$.
- Those 4 wasted bytes from $A + 20$ to $A + 23$ at the end also represent padding.
  - Even this final padding is a part of a binary representation of `X`.
  - $\Rightarrow$ These 4 bytes are part of the storage for each object of type `X`.
- *Proof:*

```
std::cout << sizeof(X);  // prints out 24
```

- $\Rightarrow$ Storage of `X` consists of 17 data bytes and 7 padding bytes.

Intermezzo — padding (*cont.*)

- *C++ rule:*
  - Class member variables with same access rights (*private*, *protected*, *public*) must be stored at growing addresses in order of their declaration.
  - ⇒ Implementation may not reorder their placement in the storage of the class object.
- ⇒ Padding may be sometimes reduced by "better" order of (non-static) member variable declarations.
- *Example:*

```cpp
struct Y {
  char  c;
  int   i;
  bool  b;
  float f;
};
```

```cpp
struct Z {
  char  c;
  bool  b;
  int   i;
  float f;
};
```

```cpp
std::cout << sizeof(Y);  // prints out 16
```

```cpp
std::cout << sizeof(Z);  // prints out 12
```

Static *vs* dynamic allocations

- *Summary* — in comparison with static allocations, allocation of storage for dynamically-allocated objects:
  - is much slower,
  - consumes much more memory.
- Efficiency with *collection of objects*:
  - = objects stored in *dynamic data structures*.
  - C++ standard library defines different types of such data structures called *containers*.
  - Comparison of containers with respect to the number of required allocations for insertion of n objects:
    - `std::vector` without reserving space $\rightarrow \log(n)$,
    - `std::vector` with reserving space $\rightarrow 1$,
    - `std::list` $\rightarrow n$,
    - `std::set` or `std::map` $\rightarrow n$,
    - `std::unordered_set` or std::unordered_map $\rightarrow n + \log(n)$.

Static vs dynamic allocations (*cont.*)

- *Experimental evaluation:*
  - Comparison of time and memory consumption when the same number of objects is inserted into different C++ library containers.
  - Measurements are *normalized* to the case where a vector was used without reserving memory.

container	time	memory (RSS)
std::vector "as-is" (*baseline*)	1	1
std::vector + reserve	2.8× faster	1.3× lower
std::list	17× slower	5.9× higher
std::set	95× slower	8.9× higher
std::unordered_set	46× slower	7.4× higher

- *Analysis:*
  - Vector is so efficient since it allocates storage for multiple objects (its elements) at once.
  - All other containers are *node-based* — each new element requires allocation of a separate storage for a single node, in which it is then stored.

Static vs dynamic allocations (*cont.*)

- Vector stores its elements contiguously in memory.
- Nodes are placed in memory at unrelated locations.
- $\Rightarrow$ Vector provides efficient element access.
- *Vector* vs *set/map*:
  - *Set/map* = binary search tree $\Rightarrow O(\log(n))$ lookup time.
  - Sorted vector with binary search $\Rightarrow O(\log(n))$ lookup time as well.

container	insertion (+sort)	lookup
std::vector "as-is" (*baseline*) + sorting	1	1
std::vector + reserve + sorting	1.2× faster	1
std::set	16× slower	2.1× slower
std::unordered_set	7.3× slower	4.6× faster

- Fastest lookup provides *unordered set* = *hash table*.
  - $\Rightarrow O(1)$ lookup time.
  - At a price of significantly slower data structure construction and higher memory consumption.

Placement new

- Up to now, all mechanisms we have seen for both static and dynamic object allocations automatically "attached":
  - storage allocations together with object initialization,
- and:
  - object destruction together with storage deallocation.
- Vector container:
  - Allocates storage for multiple objects (its elements) at once.
  - $\Rightarrow$ Needs to "detach" storage allocation from object initialization (as well as object destruction from storage deallocation).
- How can this be done in C++?
  - For storage (de)allocation alone, we have (de)allocation functions `operator new` and `operator delete`.
  - $\Rightarrow$ What we need is to initialize object in this storage (*uninitialized memory*), and — in the end — to destruct it.

Placement new (*cont.*)

- Vector — highly complex data structure.
- *Simpler case* for the sake of explanation:
  - Optional object existence — object that optionally may or may not exist at runtime.
- In case of dynamically-allocated object, the solution is simple:
  - A pointer itself can contain information whether the object exist or not:

```
int* pi = (some_condition == true) ? new int(1) : nullptr;
```

- Or, better:

```
std::unique_ptr<int> pi = (some_condition == true) ? new int(1) : nullptr;
```

- But what if we want to achieve the same without the overhead of dynamic storage allocation?
- ⇒ We need optional object initialization in a statically-allocated storage.

Placement new (*cont.*)

- Class wrapper for such an optionally existing object:

```
class optional_int {
  ??? buffer_[ ??? ];
  bool exist_;
public:
  optional_int() : exists_(false) { }
  optional_int(int i) : exists_(true) { ??? }
  operator int&() { return ???; }
};
```

- Storage for optional object is a class member variable.
  - ⇒ It is a part of the storage of the class object itself.
  - ⇒ No need to dynamically allocate it.
- This storage needs to be:
  - Large enough to hold an object of type `int` ⇒ `sizeof(int)` bytes long.
  - Aligned suitably to hold an object of type `int` ⇒ `alignof(int)` byte-aligned.

Placement new (*cont.*)

- Such storage can be provided as a buffer of a form of an array, which:
  1) is aligned to `alignof(int)`-byte address,
  2) has type of elements `unsigned char` or `std::byte` (since C++17),
  3) has `sizeof(int)` elements.

```
class optional_int {
  alignas(int) unsigned char buffer_[ sizeof(int) ];
  bool exist_;
public:
  optional_int() : exists_(false) { }
  optional_int(int i) : exists_(true) { ??? }
  operator int&() { return ???; }
};
```

- Such a storage is now suitable to hold an object of type `int`.
- Next, we need to:
  - Construct (=initialize) such an object in this storage (buffer).
  - Provide access to it (to allow its update and reading).

Placement new (*cont.*)

- Initialization of an object in existing storage = *placement new*.
  - Another form of new.
  - Basically, it is a new expression with an additionally provided pointer as its “argument”.

```
class optional_int {
  alignas(int) unsigned char buffer_[ sizeof(int) ];
  bool exist_;
public:
  optional_int() : exists_(false) { }
  optional_int(int i) : exists_(true) { new (buffer_) int(i); }
  operator int&() { return ???; }
};
```

- Placement new initializes an object:
  - in memory at the address passed as a pointer,
  - of the specified type,
  - by initialization expression the same way as for “ordinary” (non-placement) new expression.

Placement new (*cont.*)

- Object access:
  - There is no direct access (through some name/identifier).
  - But we know, where it is stored ⇒ we can derive its pointer (a *pointer-to-this-object*).
  - `buffer_` is an array ⇒ is implicitly convertible to a `pointer-to-unsigned char`.
  - We need a pointer-to-`int`.
  - ⇒ *Solution* = cast (conversion) of these pointers:

```
class optional_int {
  alignas(int) unsigned char buffer_[ sizeof(int) ];
  bool exist_;
  int* ptr() { return reinterpret_cast<int*>(buffer_); }  // helper function
public:
  optional_int() : exists_(false) { }
  optional_int(int i) : exists_(true) { new (buffer_) int(i); }
  operator int&() { return *ptr(); }
};
```

- *Note:* since the `operator int&()` returns a reference to the stored object, the casted pointer is dereferenced.

Placement new (*cont.*)

- Object destruction:
  - *Placement new* initializes an object — starts its lifetime.
  - **How to end this lifetime?**
  - Is there any "*placement delete*"? No.
- Two different cases:
  1) *Class types* — objects lifetime needs to be ended explicitly by calling their destructors.
  2) *Non-class types* — objects lifetime ends automatically when their storage is deallocated.
- `int` belongs to ad 2)
  - ⇒ No explicit action is needed for destruction of the object initialized by *placement new*.
  - ⇒ The class `optional_int` does not require manual definition of a destructor (it is *auto-generated*).
- What about ad 1) cases?

Placement new (*cont.*)

- Class wrapper for such an optionally existing class object:

```
struct X {
  X(int);
  ~X();
};

class optional_X {
  alignas(X) unsigned char buffer_[ sizeof(X) ];
  bool exist_;
  X* ptr() { return reinterpret_cast<X*>(buffer_); }
public:
  optional_X() : exists_(false) { }
  optional_X(int i) : exists_(true) { new (buffer_) X(i) }
  operator X&() { return *ptr(); }
  ~X() { if (exist_) ptr()->~X(); }
};
```

- Placement new initializes an object of type X ⇒ it calls its constructor.
- If the object was (optionally) initialized, then must be explicitly destructed by calling its destructor.
  - Destructor can be called as any other member function.

Placement new (*cont.*)

- Comparison of *non-class* (`int`; *left*) and *class* (X; *right*) cases:

```
optional_int::optional_int(int):
  mov   BYTE PTR [rdi+4], 1
  mov   DWORD PTR [rdi], esi
  ret
optional_int::~optional_int():
  ret
```

```
optional_X::optional_X(int) :
  mov   BYTE PTR [rdi+1], 1
  jmp   X::X(int)
optional_X::~optional_X() :
  cmp   BYTE PTR [rdi+1], 0
  jne   .L10
  ret
.L10:
  jmp   X::~X()
```

1. *Non-class case:*
   - *Initialization* of an object of type `int` requires just setting its memory representation to a desired value (value of constructor argument passed in the `esi` register).
   - *Destruction* does not require any action.
2. *Class case:*
   - *Initialization* of an object of type X requires calling its constructor.
   - *Destruction* requires calling its destructor.

Placement new (*cont.*)

- Problem with explicit destructor call:

```
class optional_string {
  alignas(std::string) unsigned char buffer_[ sizeof(std::string) ];
  bool exist_;
  std::string* ptr() { return reinterpret_cast<std::string*>(buffer_); }
public:
  optional_string() : exists_(false) { }
  optional_string(char* s) : exists_(true) { new (buffer_) std::string(s) }
  operator std::string&() { return *ptr(); }
  ~optional_string() { if (exist_) ptr()->~string(); }  // compilation error
};
```

```
error: expected class-name before '(' token
  ~optional_string() { if (exist_) ptr()->~string(); }
```

- `std::string` = a type alias for an instance of `std::basic_string<char>` class template.
- ⇒ There is no destructor named `~string()`.
- ⇒ The destructor is called `~basic_string()`.

```
  ~X() { if (exist_) operator X&().~basic_string(); }  // OK
```

Placement new (*cont.*)

- Generally, this is a problem, if we do not know whether some type is a type name or a type alias.
- First possible solution — templates:

```
template <typename T>
class optional {
  alignas(T) unsigned char buffer_[ sizeof(T) ];
  bool exist_;
  T* ptr() { return reinterpret_cast<T*>(buffer_); }
public:
  optional() : exists_(false) { }

  ??? // constructor – to be explained later

  operator T&() { return *reinterpret_cast<T*>(buffer_); }
  ~optional() { if (exist_) operator ptr()->~T(); }
};
```

- Generic solution — will work for any type (template argument).
- For *class types*, correct destructor name is “encoded” in `~T()` call;
- For *non-class types*, there is no destructor, such as `~int()`.
  - In such case, `~T()` call is valid, but will have no effect.
  - This mechanism is called *pseudo-destructor call* and allows us to write unified generic code where objects need to be explicitly destructed.

Teaser — perfect forwarding

- How to write a constructor of `optional`?
- Constructors of `T` may take any number of arguments having different types and different value categories.
- ⇒ We need a constructor of `optional` class that will:
  - take any number of argument of any types and any value categories,
  - and “pass” them *as-they-are* to the constructor of `T`.
- This can be solved by combining three fundamental modern C++ techniques:
  - *variadic templates*,
  - *forwarding references*,
  - `std::forward` *function*,
- which is together called *perfect forwarding*.
- *More: later lectures.*

```
template <typename T>
class optional {
  ...

  template <typename Ts...>
  optional(Ts&&... args) : exists_(true)
  {
    new (buffer_) T(std::forward<Ts>(args)...)
  }

  ...
};
```

Placement new (*cont.*)

- Second possible solution: `std::destroy_at()` library function:

```
~optional_string() { if (exist_) std::destroy_at(ptr()); }
```

- `std::destroy_at()` is a function template.
- It derives the type name (template parameter `T`) from the type of the passed pointer (*template argument deduction*).
  - Internally, it calls it destructor as `~T()`.
- *Advantage* — the code is more readable and explicit.
- Counterpart of `std::destroy_at` is `std::construct_at`:

```
optional_string(char* s) : exists_(true) {
  std::construct_at<std::string>(ptr(), s);
}
```

- Internally, it constructs an object with *placement new*:
  - its template argument represents a type of the constructed object,
  - its first function argument is a pointer to the storage,
  - all other function arguments are perfectly forwarded to the initializer.
- *Availability:* `std::destroy_at` C++17, `std::construct_at` C++20.

Placement new (*cont.*)

- Using standard library improves readability and under-standability of code; *seen examples*:
  - *placement new* vs `std::construct_at`,
  - explicit destructor call vs `std::destroy_at`.
- Another useful replacement from the standard library for our `optimal` class — `std::aligned_storage`.
  - Class (struct) template, which member type "type" provides a type of storage that is aligned to the desired value and have desired length.
  - Length and alignment are specified as template arguments.

```
alignas(T) unsigned char buffer_[ sizeof(T) ];
```

- can be replaced by:

```
typename std::aligned_storage<sizeof(T), alignof(T)>::type buffer_;
```

- or, since C++14, by:

```
std::aligned_storage_t<sizeof(T), alignof(T)>::type buffer_;  // shortcut
```

Placement new (*cont.*)

- `optional` class in C++20 with library entities:

```
template <typename T>
class optional {
  std::aligned_storage_t<sizeof(T), alignof(T)> buffer_;
  bool exist_;
  T* ptr() { return reinterpret_cast<T*>(buffer_); }
public:
  optional() : exists_(false) { }

  template<typename... Ts>
  optional(Ts&&... args) : exist_(true) {
    std::construct_at<T>(ptr(), std::forward<Ts>(args)...); }

  operator T&() { return *ptr(); }
  ~optional() { if (exist_) std::destroy_at(ptr()); }
};
```

- Such a class template in a more sophisticated form is available in the C++ standard library itself since C++17 as `std::optional<T>`.
- *Common use case* — optional returning an object from a function:

```
std::string f1();    // (1) return object always

std::string* f2();   // (2) may return optionally, but requires heap allocation

std::optional<std::string> f3();  // (3) may return optionally and...
                                  // ...does not require heap allocation
```

Placement new (*cont.*)

- Drawback of (3) *vs* (1):
  - An optional class needs a flag with the information about whether it does or does not "hold" an object.
  - This flag, for example, controls destruction.

```
template <typename T>
class optional {
  bool exist_;
  ...
  optional() : exists_(false) { }
  template<typename... Ts> optional(Ts&&... args) : exist_(true) { ... }
  ~optional() { if (exist_) std::destroy_at(ptr()); }
};
```

  - Boolean flag generally requires only 1 byte.
  - However, due to alignment requirements, there may be some padding introduced.
- Flag + padding represent "housekeeping" memory overhead:

```
std::cout << sizeof( std::string );                  // prints out 32
std::cout << sizeof( std::optional<std::string> );   // prints out 40 (+1B flag, +7B padding)
```

“Owning” objects — “owners”

- An object of type `std::optional<T>` — a “*wrapper*” that (optionally) “*owns/manages*” an object of type `T`.
  - ⇒ The “*content*” of `std::optional<T>` is an (optional) ownership of an object of type `T`.
- The same holds, for instance, for an object of type `std::unique_ptr<T>`.

```
std::optional< std::string > os( "..." );
std::unique_ptr< std::string > us( new std::string( "..." ) );
```

- *Further assumption* — stack/heap-based C++ implementation.
- Difference between `optional` and `unique_ptr`:
- `unique_ptr` — the storage of the owned object is **allocated dynamically**.
  - ⇒ The owned object itself is stored on the heap.
- `optimal` — the storage of the owned object is “**included**” in the storage of the owning (`optimal`) object itself.
  - ⇒ For instance, if the `optimal` object is stored on the stack, so is also the owned object.

“Owners” (*cont.*)

- *Physical ownership* — owned object is stored in the storage of its owner (= *included* storage).
- *Logical ownership* — owner takes care about owned object(s) (manages lifetime etc.).
- `std::optional<T>` — both *physical* and *logical* ownership:
  - Object is, for example, initialized and destructed by its `optional` owner.
  - Included storage — *example with GCC/x86_64*:

```
std::optional<int> oi(1);

std::cout << &oi;           // address of owner: "0x7ffc2e7146a8"
std::cout << &oi.value();   // address of owned: "0x7ffc2e7146a8"
```

  - ⇒ The owner `oi` is stored on the *stack*, so is the owned object.
  - ⇒ Storage size of the `optional` owner depends on the storage size of the owned object.

```
std::cout << sizeof( std::optional< int > );           // "8"
std::cout << sizeof( std::optional< std::string > );  // "40"
```

“Owners” (*cont.*)

- `std::unique_ptr<T>` — *logical* ownership only:
  - Object is, for example, destructed by its `unique_ptr` owner.
  - Dynamically allocated storage:

```
std::unique_ptr<int> ui(new int(1));
std::cout << &ui;   // address of owner: "0x7ffe276c8450"
std::cout << &*ui;  // address of owned: "0x23faec0"
```

  - ⇒ The owner `ui` is stored on the *stack*, but the owned object is stored on the *heap*.
  - ⇒ Storage size of the `unique_ptr` owner does not depend on the storage size of the owned object.

```
std::cout << sizeof( std::unique_ptr< int > );          // "8"
std::cout << sizeof( std::unique_ptr< std::string > );  // "8"
```

- *Note:* `std::unique_ptr<T>` is typically implemented as a wrapper of the raw pointer member variable of type `T*`.

“Owners” (*cont.*)

- *C++ standard library* provides multiple generic “owner” types.
- Some *single-object owners*…
  - …with *included storage*:
    - `std::optional<T>`, `std::variant<T1,T2,…>`;
  - …with *dynamically-allocated storage*:
    - `std::unique_ptr<T>`, `std::shared_ptr<T>`, `std::any`.
- Some *multiple-object owners*…
  - …with *included storage*:
    - `std::pair<T1,T2>`, `std::tuple<T1,T2,…>`, `std::array<T,N>`;
  - …with *dynamically-allocated storage*:
    - *dynamic containers* — `std::vector<T>`, `std::list<T>`, `std::set<T>`,…
- What about `std::string`?
  - An object of type `std::string` can handle/own a *string of characters* of any length, generally unknown at compile time.
    - ⇒? Must the *string of characters* be stored in dynamically-allocated storage?

Owners — `std::string` case

- *Experiment — Clang/x86_64/libc++:*

```
std::string s("...");
std::cout << &s              // address of the string object:          "0x7ffef075e4c0"
  << (void*)s.data();  // address of the string-of-characters: "0x7ffef075e4c1"
```

- *Observation:*
  - The owner s is stored on the stack.
  - The string of characters "..." managed/owned by s is stored also on the stack.
  - Namely, it is stored in the *included storage* of its owner s.
- *Similar experiment:*

```
std::string s("... some different string ...");
std::cout << &s              // address of the string object:          "0x7fffd772d280"
  << (void*)s.data();  // address of the string-of-characters: "0x18bf2a0"
```

- *Observation:*
  - Now, the string of characters "`... some different string ...`" is stored on the heap (in a *dynamically allocated storage*).
- *Explanation* — see next presentation.