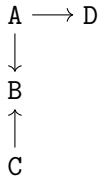
Concepts of C++ Programming

Tuesday, February 04, 2025 14:30 – 15:15

Problem 1: Struct Layout and Inheritance *(4 pts)*

Given a (visual) representation of an inheritance graph, desired sizes, and implementations of each class/struct, recreate the inheritance structure and struct sizes only by adding inheritance relations.
Note: `X` $\longrightarrow$ `Y` Indicates that Y is derived from X;



a) *(2 pts)* Adjust the structs so that they have the expected inheritance structure

b) *(2 pts)* Reorder the members of each class to make them as small as possible (*Do not remove members or change their types!*)

Problem 2: Vector *(35 pts)*

Implement a subset of the standard `std::vector<T, Alloc>`. As a reminder, a vector represents a continuous storage of items. It is implemented by keeping track of the current number of items stored in it (*size*) and the number of potentially insertable items within the current allocation (*capacity*). When inserting into a vector where size equals capacity, the vector has to reallocate its internal storage (doubling the previous capacity), transfer the old elements to their new location, and append the new element. For this exercise, ensure an empty vector has capacity 0 and then grows to 1 on first insertion (doubling for succeeding relocations).
Besides the template argument for the item type, the vector is further specialized on the used allocator (defaulting to `std::allocator<T>`). An instance of the allocator can be passed upon construction and should be used for all allocations and deallocations. Using `new` and `delete` for the tasks is possible, but it will lose you 3 points on the total exercise.
Whenever transferring items from one location to another, prefer moving the object when possible. If an object is not movable but copyable, fall back to copying the objects. Using the **vector** header of the STL is forbidden!

1. **Fundamentals** (total 2 pts): Define the `vector::vector<T, Alloc>` class with a default constructor and a constructor that takes an `std::initializer_list<T>` of elements to initialize the vector with (by copy). Each should have an optional argument of type `Alloc`. If not explicitly given, the allocator should be default-constructed.
   Important: This is a prerequisite for all the following features. Failing to implement these two constructors will result in the entire task receiving zero points.

2. **Constructors** (total 8 pts):
   - Define a copy constructor that uses `T`'s copy constructor to copy all elements from the source into a new vector. The new vector allocates exactly as much memory as required to hold all elements. Also, define an analogous copy assignment operator.
   - Define a move constructor that should not move or copy any elements but take ownership of the other vector's contents. Also, define an analogous move assignment operator.
   - Define a static function `with_capacity(size_t capacity)` that creates an empty vector but with preallocated space for the given number of `capacity` elements.
   - Define a constructor that takes two parameters, `size_t count, const T& value`, and creates a vector by copying `value` `count` times. The resulting capacity and size should be exactly `count`.

3. **Accessors** (total 7 pts):
   - Define member types `value_type` equal to `T` and `allocator_type` equal to `Alloc`.
   - Define `empty`, `size`, `capacity` with the obvious semantics.
   - Define the `operator [](size_t idx)` that accesses the element at a given index with no bounds checking

- Define the `at(size_t idx)` method that accesses the element at a given index or throws an `std::out_of_range` exception when out of bounds.
- Define the `data` method returning the pointer to the inner `T` data owned by the vector
- Define the `front` method that returns a reference to the first element, and `back` that returns a reference to the last element (no checks for non-empty vector).

Note: All accessors should be `const` and `noexcept` where it makes sense. If the accessor returns a pointer or a reference, ensure there is both a `const` and non-`const` overload.

4. **Modification** (total 7 pts):
   - Define two `push_back` methods, one that adds an element at the end of the vector by copying and the other by moving it.
   - Define `emplace_back` that creates an element in-place at the end of the vector by perfect-forwarding the arguments to `T`'s constructor. Return a reference to the newly created object.
   - Define `pop_back` that removes the last element without affecting capacity.
   - Define `reserve(size_t capacity)` that assures capacity is at least a given value; does nothing if capacity is already greater or equal to the parameter - increases the capacity to the provided value otherwise.
   - Define `clear` that removes all elements without affecting capacity.
5. **Iterators** (total 6 pts). Define the `iterator` and `const_iterator` member types. Define the functions `begin()` and `end()` that return iterators to the vector, both in the const and non-const version. The number of points depends on the functionality implemented for the iterators, represented by the progression of standard iterator traits:
   - `std::input_iterator`;
   - `std::forward_iterator`;
   - `std::bidirectional_iterator`;
   - `std::random_access_iterator`;
   - Add the `reverse_iterator` and `const_reverse_iterator` member types, and the `rbegin` and `rend` returning reverse iterators;
   - Ensure all iterators have a size of at most 8 bytes.
6. **Size** (total 2 pts): The size of a vector object that is specialized with the default allocator should be as small as possible.
   - (1 pt) size is less than or equal to 32 bytes
   - (1 pt) size is less than or equal to 24 bytes

   Note: The default allocator is a structure with no fields. Numbers might be different for non-empty allocator types.

Problem 3: Memory Allocations *(4 pts)*

Given the following code snippet.

```
inline int exercise()
{
  int* a = new int;
  int* b = new int;

  delete a;
  delete b;

  return -1;
}
```

a) *(1 pts)* What statement about it is correct? Provide your answer by returning the corresponding number from the `exercise` function in `memory-allocation/exercise.hpp`.
Example: if there was no problem with the code (*answer 1*), change the line to return `1` instead of `-1`.

1. There is no problem with the shown code.
2. The code is lacking null assignments after the `delete` as `C++` requires the user to set a pointer to `nullptr` after deleting its content.
3. The dynamic memory allocation is unnecessary, so the compiler will reject that code.
4. Because `new` is potentially throwing, we might leak memory as the `delete` lines might never be called.

b) *(3 pts)* Fix the identified issues in the code or leave it as is if you did not identify any. Do not change the observable behavior - in particular, keep two memory allocations and deallocations.