Ejercicio sobre **`std::vector`** y move semantics

El objetivo de este ejercicio es entender el funcionamiento de la clase **`std::vector`** y la importancia de la construcción/**`operator=`** de movimiento.

Descarga el archivo **`stdvec_examples.cpp`** desde el campus y añadelo a un proyecto nuevo. Notase que usamos variables globales para contar el número de veces que ejecutamos **`new`**, **`delete`**, las constructoras, etc. El método **`print_stats()`** escribe las estadísticas y él método **`reset_stats()`** las resetea.

La diferencia entre `push_back` y `emplace_back`.

Ya sabemos que el método **`push_back`** añade nuevo elemento al array copiado (o moviendo) el objeto que recibe como parámetro. El método **`emplace_back`** recibe los parámetros que hay que pasar a la constructora y crea un nuevo objeto.

A continuación tenemos varios ejemplos que usan **`push_back`** y **`emplace_back`**

`std::vector<A> v;`
`v.push_back(A(1));`
	`std::vector<A> v;`
	`A x(1);`
	`v.push_back(x);`
		`std::vector<A> v;`
		`A x(1);`
		`v.push_back(std::move(x));`
			`std::vector<A> v;`
			
			`v.emplace_back(1);`

1. Ejecutarlos e inspeccionar el resultado de **`print_stats`** para entender la diferencia entre ellos.
2. Quita la constructora de movimiento de la clase A (ponla en comentario), y ejecutarlos e inspeccionar el resultado de **`print_stats`** comparándolos con los del apartado anterior.

Lo que cuesta redimensionar la memoria.

La clase **`std::vector`** tiene memoria ya reservada para **`N`** objetos y cada vez que añadimos un objeto lo almacena en esa memoria. Esto es muy importante para evitar el uso excesivo de **`new`**/**`delete`** y para mantener los objetos en un memoria contigua que es super importante para el uso de la memoria caché.

Cuando se rellena todas las posiciones de la memoria reservado, reserva una memoria para **`2*N`** objetos y copia los objetos que están en la memoria anterior a la nueva (creando nuevos objetos usando la constructora de copia o movimiento). Por defecto `std::vector` usa **`N=0`** (la primera vez redimensiona a **`N=1`** no a **`2*N`**).

El siguiente cçodigo añade 10 elementos a un **`std::vector`**

```
std::vector<A> v;
for (int i = 0; i < 10; i++) {
	std::cout << "# adding " << i << " using emblace_back" << std::endl;
	if (v.capacity() == v.size()) {
		std::cout << "# The vector is about to be resized!" << std::endl;
	}
	v.emplace_back(i);
}
```

Estudia la diferencia, inspeccionando **`print_stats`** y los mensajes que imprime la clase `A` para las siguientes configuraciones:

1. Ejecutar el código como está arriba sin/con la constructora de movimiento
2. Añadir **`v.reserve(20)`** antes del bucle y ejecutalo (con la constructora de movimiento).

En lugar de quitar la constructora de movimiento, intenta solo quitar la keyword **`noexcept`** de dicha constructora. En este caso **`std::vector`** no usa esta constructora por el riesgo de que salte una excepción mientras está copiando los elementos a la nueva memoria. Esta keyword garantiza que eso no puede ocurrir (si lanzas una excepción en dicha constructora el código no compila).

Borrar usando std::find_if y std::remove_if.

El siguiente código borra todos los elementos pares del vector:

```
std::vector<A> v;
v.reserve(20);
for (int i = 0; i < 10; i++) v.emplace_back(i);
auto pred = [](A &x) { return x % 2 == 0; };
std::vector<A>::iterator it;
while ((it = std::find_if(v.begin(), v.end(), pred)) != v.end()) {
        std::cout << "# erasing " << *it << std::endl;
        v.erase(it);
}
std::cout << "array after deletion " << std::endl;
print_vector(v);
```

El problema es que en cada iteración desplaza todos los elementos que están a la derecha del elemento que se ha borrado una posición hacia la izquierda. La complejidad es cuadrática. Para observar este comportamiento, ejecuta el código e inspeccionando **`print_stats`** y los mensajes que imprime la clase `A`.

Para resolver este problema podemos usar **`std::remove_if`** que desplaza (usando **`operator=`**) todos los elementos que cumplen la condición al final del vector y devuelve un iterador al inicio de este segmento, usando un solo recorrido sobre el vector. Ahora podemos borrar todos los elementos de ese segmento sin desplazar nada.

El siguiente código usa **`std::remove_if`** para borrar todos los elementos pares:

```
std::vector<A> v;
v.reserve(20);
for (int i = 0; i < 10; i++) v.emplace_back(i);
auto pred = [](A &x) {
        return x % 2 == 0;
};
std::vector<A>::iterator it = std::remove_if(v.begin(), v.end(), pred);
v.erase(it, v.end());
std::cout << "array after deletion " << std::endl;
print_vector(v);
```

Para observar este comportamiento, ejecuta el código e inspeccionando **`print_stats`** y los mensajes que imprime la clase **`A`**, sin y con **`operador=`** de movimiento (simplemente ponlo en comentario).

El código de **`std::remove_if`** como aparece en las librerías es el siguiente:

```
template <class _ForwardIterator, class _Predicate>
_LIBCPP_NODISCARD_EXT _LIBCPP_CONSTEXPR_AFTER_CXX17 _ForwardIterator
remove_if(_ForwardIterator __first, _ForwardIterator __last, _Predicate __pred)
{
    __first = _VSTD::find_if<_ForwardIterator, typename
add_lvalue_reference<_Predicate>::type>
                            (__first, __last, __pred);
    if (__first != __last)
    {
        _ForwardIterator __i = __first;
        while (++__i != __last)
        {
            if (!__pred(*__i))
            {
                *__first = _VSTD::move(*__i);
                ++__first;
            }
        }
    }
    return __first;
}
```

La primera llamada a **`find_if`** encuentra el primer elemento que cumple la condición del predicado. Ejecutalo (a mano!) para el ejemplo que usamos arriba, y indica para qué posiciones del array ejecuta la instrucción **`*__first = _VSTD::move(*__i)`**, es decir a qué posiciones **`__first`** y **`__i`** corresponden en cada ejecución de esta instrucción.