1. Instancie contenedores **`std::list<int>`**, **`std::deque<int>`** y **`std::vector<int>`**. Agrégueles enteros con valor random. Ordénelos con **`std::sort`** midiendo el tiempo necesario para realizar tal operación. Repita el procedimiento cambiando la cantidad de elementos en los contenedores. Compruebe el orden del algoritmo para cada contenedor. (1)
2. Sobre uno de los contenedores del ejercicio 1, aplique la función **`for_each`** que se encargue de duplicar cada uno de los valores contenidos.
3. Utilizando el contenedor resultado del ejercicio 2, aplique la función **`all_of`** para verificar que todos los valores contenidos son pares.
4. Instancie contenedores **`std::set<int>`** y **`std::unordered_set<int>`**. Agrégueles enteros con valor random iguales a ambos contenedores. Ejercite múltiples búsquedas sobre los contenedores midiendo el tiempo necesario para realizar tales operaciones. ¿Detecta diferencias de performance? ¿Cómo lo justifica? (1)
5. Instancie una **`std::list<int> lista`**, inserte 5 enteros, por ejemplo la secuencia **`{1, 2, 3, 4, 5}`**. Obtenga un referencia (iterador) al comienzo de la secuencia a través de **`lista.begin()`**. Experimente distintas operaciones que incluyan incrementar o decrementar el iterador (++ y --) y el método **`insert`** para ir insertando nuevos elementos a esa lista, utilizando el iterador como posición de inserción. Vaya observando los resultados imprimiendo la lista.
   Hint: http://www.cplusplus.com/reference/list/list/insert/
6. Una cola (queue en inglés) es una estructura de datos que permite almacenar y recuperar datos, siendo la semántica de acceso a sus elementos del tipo **FIFO** (del inglés **F**irst **I**n, **F**irst **O**ut, «primero en entrar, primero en salir»).
   Para el manejo de los datos cuenta con dos operaciones básicas: **`put`**, que coloca un nuevo elemento en la cola, y su operación inversa, **`get`**, que retira el elemento más antiguo. Puede existir además una operación para recuperar la cantidad de elementos en ella (**`size()`**).
   Se puede ver como una estructura lineal, en donde los elementos son ingresados por un extremo y retirados por el otro.
   Se solicita:
   - Definir el tipo de ***`CONTAINER`*** más conveniente de acuerdo al tipo de operaciones que se llevaran a cabo sobre él.
   - Implementar los métodos de la interface básica del UDT **`Queue`** para almacenar entidades del tipo T:

```
template <class T>
class Queue {
    public:
        void put(T e);          // coloca un nuevo elemento en la cola
        T get();                // retira el elemento más antiguo de la cola
        unsigned int size();    // retorna la cantida de elementos contenidos
    private:
        std::CONTAINER<T> elts; // a definir
};
```

7. Un esquema bastante común para hacer el procesamiento de “Jobs” en clusters de computadoras es el de modelar el concepto de una “cola de tareas”. Usuarios de ese cluster registran nuevos Jobs registrándolos en un planificador (**`JobManager`**) y este los coloca en una cola para que esos Jobs sean ejecutado después de que todos los Jobs anteriormente registrados sean ejecutados. En general, también existe el concepto de

Workers, como las entidades responsables de la ejecución de Jobs. Éstos se encargan de ir consumiendo la cola de Jobs e ir ejecutándolos ordenadamente.

En el código presente en **JobManager.cpp** se da una sobresimplificación del esquema que claramente resulta insuficiente para un sistema real (donde existen múltiples computadoras, CPUs, workers, etc.), pero que permite simular de una forma simple el funcionamiento de las distintas entidades.

Se solicita completar el código y probarlo con el **main** dado.

8. *(Ejercicio extra, opcional, avanzado)* Como implementaría un **ForwardList** con funcionalidad similar al **std::forward_list**? Incluya la posibilidad de obtener un iterador al contenedor (**ForwardList::Iterator**) a través de los métodos **begin()** y **end()**.

---

(1) Para medir el tiempos entre dos marcas puede utilizarse el siguiente "snippet"

```
#include <chrono>

using namespace std;
using namespace std::chrono;
...
auto t0 = high_resolution_clock::now();
// do work
// ...
auto t1 = high_resolution_clock::now();

// se imprime el tiempo que hay entre t1 y t0.
cout << duration_cast<milliseconds>(t1-t0).count() << " msec\n"
```

(2) Para suspender la ejecución de un programa durante un lapso de tiempo puede utilizarse el siguiente "snippet"

```
#include <chrono>
#include <thread>

using namespace std;
using namespace std::chrono;
...
auto t0 = high_resolution_clock::now();

// suspende la ejecución (duerme) durante 20 millisegundos
this_thread::sleep_for(milliseconds{20});

auto t1 = high_resolution_clock::now();

cout << duration_cast<microseconds>(t1-t0).count() << " usecs\n";
```