Exercise 1 (Concurrency architecture)

1. We have provided you with a template class, `MessageQueue`, that adapts the standard library std::queue.

   The `MessageQueue` class has three methods

   - `void push(const T&)` – Adds a value to the queue
   - `T pull()` – Retrieves a value from the queue. This method removes the value from the queue)
   - `bool isEmpty()` – returns true if there are no data items in the queue; false if there are data items in the queue.

   The `MesssageQueue` class will throw an exception (of type `std::range_error`) if the number of items added to the `MessageQueue` exceeds the maximum size template parameter, or an attempt is made to pull from an empty `MessageQueue`.

2. Create a class `Generator`.

   The `Generator` class must have an association (that is, a pointer) to a `MessageQueue` object, which is its output pipe.

   (Hint: Use the constructor to 'bind' the `Generator` to the `MessageQueue`)

3. Add a method, `int run()` to the `Generator` class. When called, the `run()` function must generate a random number and push it to its output `MessageQueue`.

   The function should return the generated number.

4. Create a class `LowPass`.

   The `LowPass` class must have two associations to `MessageQueue` objects – an input pipe and an output pipe.

   The `LowPass` class should hold an attribute (member variable) that represents the 'pass value' for the filter. Any value equal to, or greater-than, the filter value, should be 'filtered out'.

5. Add a method `int run()` to the `LowPass` class. When called the `LowPass` object should pull a value from its input (if there is one!).

   Values below the filter value should be pushed to the `LowPass`'s output pipe; otherwise discarded.

   The function should return the retrieved number (or -1 is the input was empty).

 © Feabhas Limited 1995-2015

6. Create a class `Display`

   The Display class must have one association to a `MessageQueue` object – its input pipe.

7. Add a method `int run()` to the `LowPass` class which, when called, retrieves a value from the input pipe (if there is one) and displays it to the console.

   The `run()` function should return the displayed value (or -1 if the input was empty)

8. Create instances of your filter objects in `main`.

   (Hint: you will need two pipes for this system)

   Connect your filter objects together with `MessageQueue` objects

   (Hint: The output from the `Generator` object must be the input to the `LowPass` object; the output of the `LowPass` object is the input to the `Display` object)

9. Create an infinite loop that calls `run()` on each filter object in turn

   (Hint: for simplicity, call the `Generator` first, then `LowPass`, then `Display`)

 © Feabhas Limited 1995-2015

Exercise 2 (Implementing concurrency)

1. Create a free function (that is, not a member of a class) `int generator_run(void* param)` to run your `Generator` object in its own thread-of-control.

   (Hint: you will need to pass the address of your `Generator` object as a parameter).

   The `generator_run()` function should call your `Generator`'s `run()` function in a `while` loop. To make the output of the program more readable you should put the thread to sleep at the end of each iteration (one second is plenty).

2. Create a task that runs the `generator_run()` function.

3. Remove the `while(true)` loop from the previous exercise.

4. Repeat part 1 for each of the other filter objects.

   (Hint: Make sure you connect all the pipe and filter objects together before creating the tasks!)

 © Feabhas Limited 1995-2015

Exercise 3 (Thread-Is-Polymorphic-Object)

1. Create a new class `Thread` that implements the *Thread-Is-Polymorphic-Object* pattern.

2. Modify your `Generator`, `LowPass` and `Display` classes so that they inherit from the `Thread` base class.

   (Hint: You will have to move the task's `while` loop inside the filter class's `run()` function)

 © Feabhas Limited 1995-2015

Exercise 4 (Thread-Runs-Polymorphic-Object)

1. Modify your `Thread` class so that it implements the *Thread-Runs-Polymorphic-Object* pattern.

2. Modify your filter classes so that they inherit from the `IRunnable` interface.

   (Hint: You should remove the `while()` loop in the filter's `run()` method, as this will now be handled by the `Thread` class's run policy)

 © Feabhas Limited 1995-2015

Exercise 5 (Mutual exclusion)

1. Create a `Mutex` class that represents a mutual exclusion primitive.

2. Modify the `MessageQueue` class so that it is thread-safe.

   Hints:

   - *Make sure you protect ALL accesses to the underlying `std::queue`.*
   - *Make sure you unlock on ALL exit paths from a function (including exceptions!)*

 © Feabhas Limited 1995-2015

Exercise 6 (Scoped-lock idiom)

1. Implement the *Scope-locked idiom* for your `Mutex` class

2. Modify the `MessageQueue` class to use a lock-guard object.

 © Feabhas Limited 1995-2015

Exercise 7 (Thread synchronisation)

1. Create a `Signal` class that represents a unilateral, persistent, consuming synchronisation primitive.

2. Modify your `MessageQueue` class so that it blocks on an empty queue.

3. Modify your filter classes so that they no longer 'busy-wait' on an empty pipe.

4. OPTIONAL – Modify your code so that the `MessageQueue` also blocks when full.

   (Hint: You will no longer need to throw the exception in this case)

 © Feabhas Limited 1995-2015

Exercise 8 (Monitors)

1. Create a class `Condition` that represents a *condition variable.*

2. Modify your `MessageQueue` class to implement the *Monitor* pattern

 © Feabhas Limited 1995-2015

Exercise 9 (Asynchronous message)

We will modify the pipe-and-filter pattern to allow our filter objects to communicate with asynchronous messages, rather than via pipes.

1. Modify the `Display` class to have a (thread-safe) `MessageQueue` as a composite member.

2. Add a new method to the `Display` class, `void show(int val)`. The body of the `show()` method should post the `int` onto the internal `MessageQueue`.

3. Add a new private method to the `Display` class, `void show_impl(int val)`. When called, this function should display the argument (that is, the same behaviour from previously)

4. Modify the Display's `run()` function so that when there is a value in the `MessageQueue`, the `show_impl()` function is called with the new value.

5. Modify the `LowPass` class so that it now has an *association* to a `Display` object, rather than to a pipe.

6. Repeat the above steps for the `LowPass` class, but this time the asynchronous method should be called `void process(int val)`; with a corresponding implementation, `void process_impl(int)`

   When the `process_impl()` method runs it should make an asynchronous call to the `Display` object, that is:

```
void LowPass::process_impl(int val)
{
  // If value is below threshold...
  //
  display->show(val); // <= 'Asynchronous call'
  ...
}
```

7. Modify the `Generator` class so that it has an association (pointer) to a `LowPass` class.

8. Modify the `Generator`'s `run()` function so that it makes an asynchronous call to the `LowPass`'s `process()` method.

 © Feabhas Limited 1995-2015

9. Modify your object construction in `main()` to bind the `Generator`, `LowPass` and `Display` objects together

   (Hint: You no longer need the `MessageQueue` pipes)

10. OPTIONAL – Modify your code so that your filter classes can support many different Asynchronous calls.

    Hints:

    - All asynchronous calls should have the same signature (eg. `void func(int)`)
    - Consider using a `struct` that holds both the pointer-to-member-function and parameter
    - To dispatch a pointer-to-member-function use

      ```
      this->*mem_fn_ptr();
      ```

 © Feabhas Limited 1995-2015