Homework 6 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

Question 1a

```
baz
bar
1
```

Question 1b

```
baz
bar
4
```

Question 1c

```
baz
bar
1
```

Question 1d

```
bar
1
```

NOTE: I'm not sure if `a` remains as 1 in the global scope in pass-by-need semantics, but I know that `baz` shouldn't be printed because the expression of `baz()` ended up not being used anywhere.

Question 2

The first implementation is using the **Optional object** return pattern for propagating errors. Consumers of this API have to remember to check after the function call if the result object contains `nullptr` instead of a valid value. This pattern works better for recoverable errors and valid results and where the error is reasonably

common. Furthermore, specifically an Optional suffices in place of a Result because there's only one error state - the element doesn't exist.

The second implementation is using the **exception** pattern for propagating errors. Consumers of this API should check after the function call if there was an error by `catch`ing a possible exception and then handling it gracefully. One upside to this approach however is that the direct caller need not be the one to handle the exception, but any caller higher in the call stack. This pattern works better for errors that are *exceptionally* rare because exceptions are regarded as *separate* from the core logic and have a significant overhead associated with them.

Either approach can be appropriate depending on the how common is the case of the array not containing the requested element. If it's reasonably common, an Optional is more suitable; if it's very rare and probably indicative of some error of critical nature, then an exception is more suitable.

Question 3

Question 3a

```
catch2
I'm done!
that's what I say
Really done!
```

In `foo(0)`:

A `range_error` is thrown, which isn't derived from `logic_error` but is derived from `runtime_error`. Thus, the inner `catch` in `foo` is ignored and execution resumes at the outer `catch`, printing `catch2` and then executing everything after the outer `try-catch` clause.

Back in `bar(0)`:

Since the exception was handled in `foo`, `foo(x)` is an exception-free call, so execution resumes in the `try` block, printing `that's what I say` and skipping over the `catch` block. Finally, the final `Really done!` is printed.

Question 3b

```
catch 1
hurray!
I'm done!
that's what I say
Really done!
```

`invalid_argument` is derived from `logic_error`. Execution continues according to similar rules described in 3a, which I'll be skipping for the rest of Question 3.

Question 3c

```
catch 1
hurray!
I'm done!
that's what I say
Really done!
```

`logic_error` is caught directly by the inner `catch` statement in `foo`. The output is identical to that in 3b.

Question 3d

```
catch 3
```

`bad_error` isn't derived from either `logic_error` or `runtime_error`, so execution is brought to `bar`'s `catch` statement, which catches all exceptions.

The final `Really done!` is not printed because `return` is used without the presence of a `finally` clause.

Question 3e

```
hurray!
I'm done!
that's what I say
Really done!
```

No error is thrown at all, so no code in `catch` blocks are run, and all other lines in and out of `try` blocks are run normally.

Question 4

Question 4a

```
template <typename T>
class Kontainer
{
public:
    Kontainer() : size(0){};
    void add(T item);
    T getMin();

private:
    T elements[100];
    size_t size;
};

template <typename T>
```

```
void Kontainer<T>::add(T item)
{
    this->elements[this->size++] = item;
}

template <typename T>
T Kontainer<T>::getMin()
{
    T *min = nullptr;
    for (size_t i = 0; i < this->size; ++i)
    {
        T *current = &this->elements[i];
        if (min == nullptr || *current < *min)
            min = current;
    }
    return *min;
}
```

Question 4b

Skipped.

Question 4c

Skipped.

Question 4d

Skipped.