Kristian Snyder

Data Structures CS2021 Section 001

Prof. Charles Zimmer, Prof. Karthik Vadambacheri

Lab 05

Writing code to handle and use exceptions, as well as creating templated and flexible classes were the focuses of this lab. Each of these concepts, when applied in large codebases and small projects alike, can dramatically increase readability and extensibility of the code.

Exceptions provide a clear and accepted code path for errors that is easy to understand and encouraged by the language. They allow others reading and using existing code to easily predict behavior caused by the exceptions due to the standardization of exceptions, which makes adding code to functions using them far easier. Additionally, it takes a lot of the work out of creating a custom error setup and making design decisions related to that when creating a new project. Adopting exceptions provides the benefits of a well-understood error model and years of consideration and optimization of the process as well as extensive use in enterprise and individual environments alike, so anyone working on a C++ project at an advanced level would welcome the use of exceptions. This makes them very attractive to learn and master.

Templating can result in overgeneralization of classes and functions—an anti-pattern wherein every function becomes a templated function just in case multiple types ever need to use it—but in many cases, can be a very useful concept to avoid code duplication and improve future refactoring when necessary. In the vein of providing base classes that serve more general functions and then derived classes that take the direction of the base in a more specific direction, templates work well to provide this level of abstraction and encourage code reuse. When there are fewer elements to override with a derived class due to templating, it makes the speed of creating reliable code quicker.

Additionally, since templates provide a syntax that is distinctive in C++, it makes any use of them obvious and signals the potential for reuse that is commonly missed when attempting to create the same signals with naming or overloading. Finally, since templating generates code at compile time, it does not result in a performance loss due to its use—although it does increase compilation duration.

*Note: At the recommendation of the instructions of the lab, I read the entire lab before beginning and created a template-based animal collection class originally instead of binding the functionality to Horse. Therefore, the differences between the code for Tasks 1-3 and Task 4 may be smaller than anticipated.*

**Task 1: Add and Remove Stable Methods**

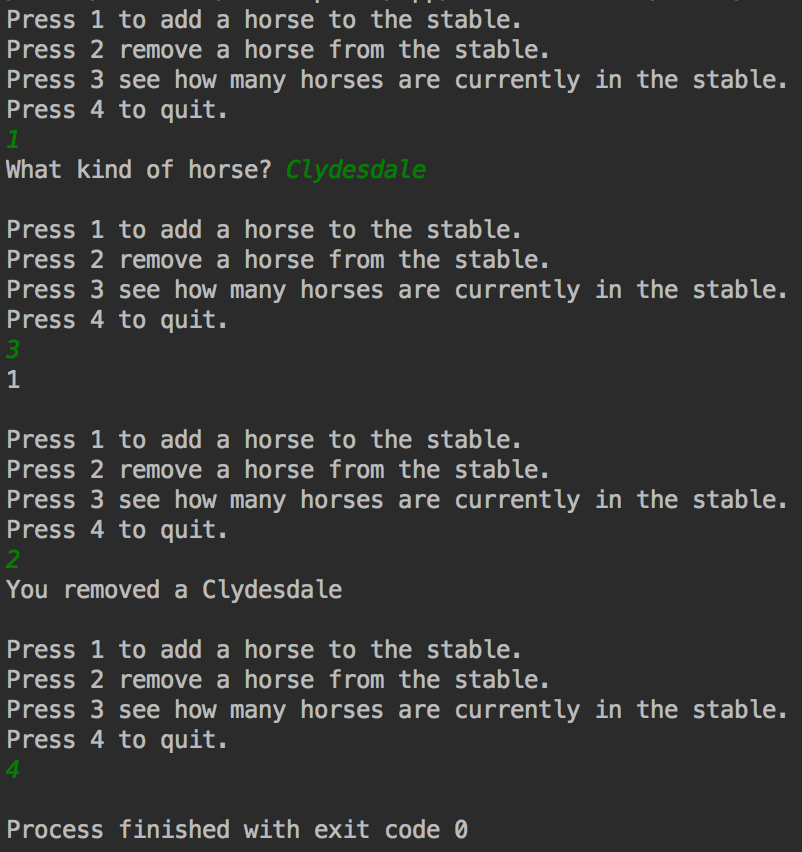
With the stable represented as an array, I decided to implement its functionality as a stack, with LIFO storage and retrieval.

The number of horses in the stable is used as the next empty index of the array; when there are no horses, the variable is 0, for three horses the variable is three. The variable always points to the next open element of the array—the exception being for a full stable, but that is out of the scope of this task.

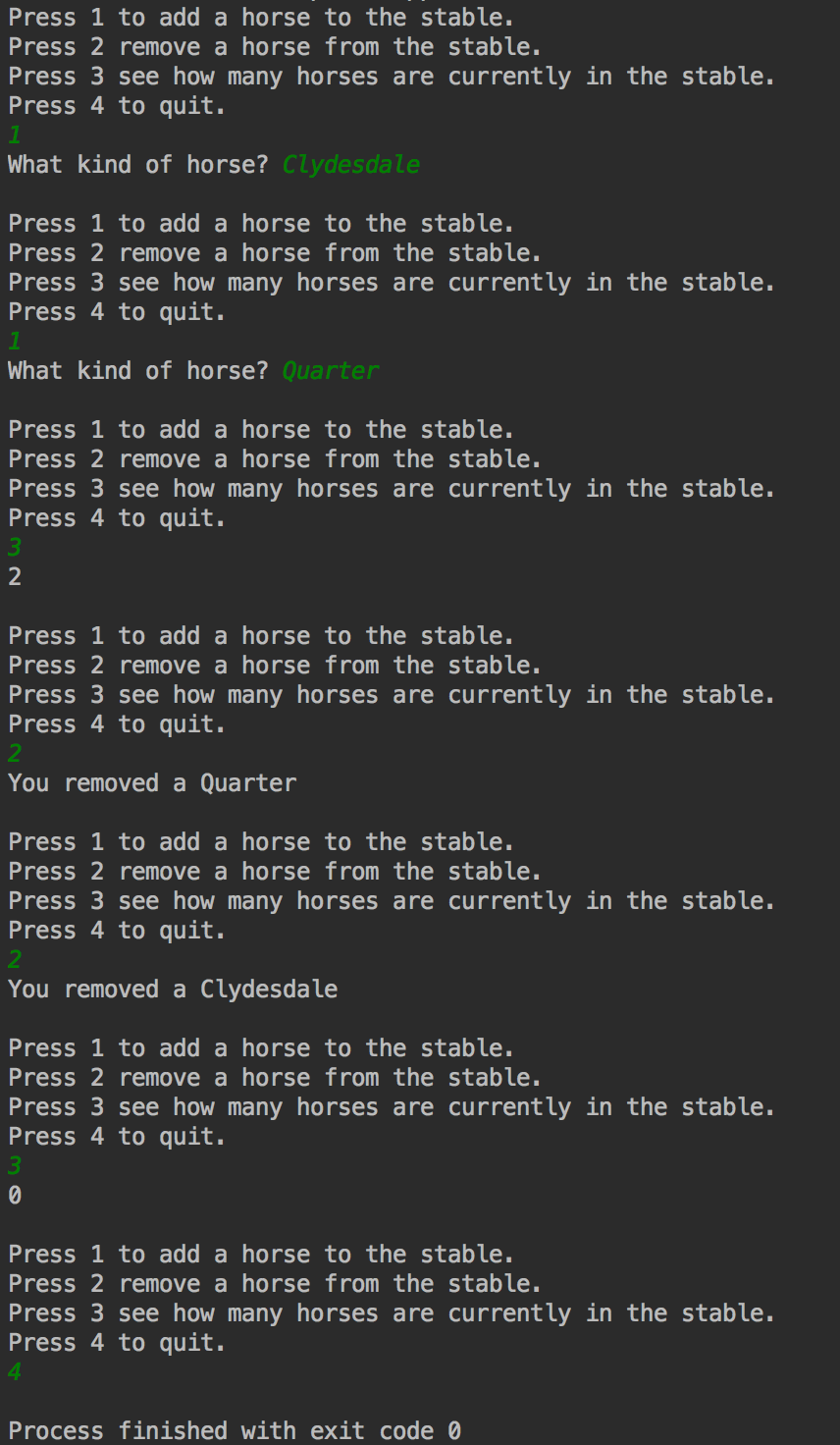
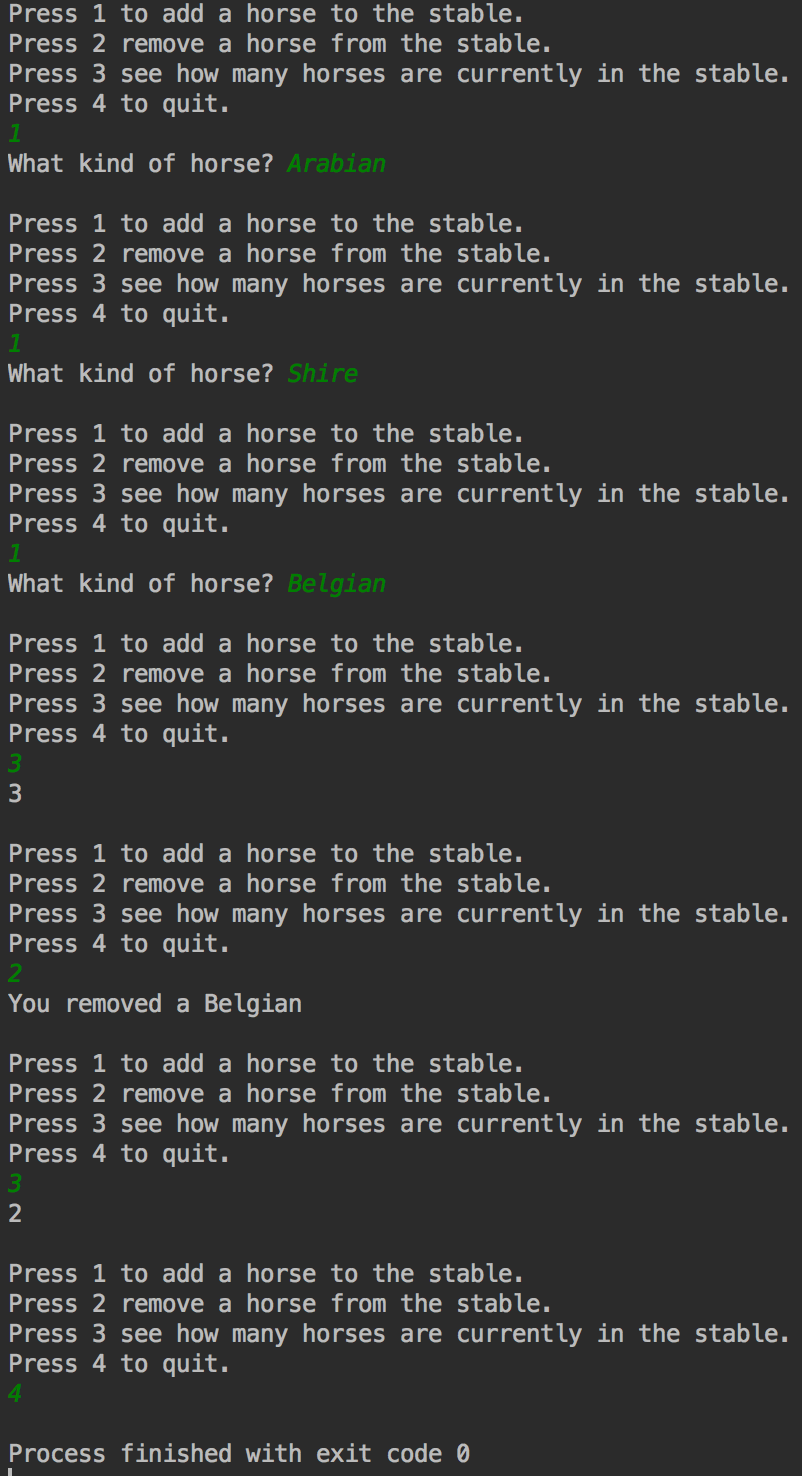
The function addAnimal(T animal) first looks at the next open spot in the array. (In a future task, checking to see if this spot is valid will be added.) Next, the animal passed into the function is placed into that spot in the stable’s array and the number of animals is incremented. This moves the next open spot one to the right for another animal to be added into (if valid).

T removeAnimal() first decrements the number of animals in the array. This serves the obvious purpose of tracking that an animal has been removed, but also, due to the number pointing to the next open spot, moves one left to the most recently added animal. This animal is then returned. Setting the element returned to NULL or deleting the spot in the array is not necessary due to all the elements being statically allocated; the array consists of objects, not pointers. (However, in task 2 this becomes necessary.)

**Task 2**

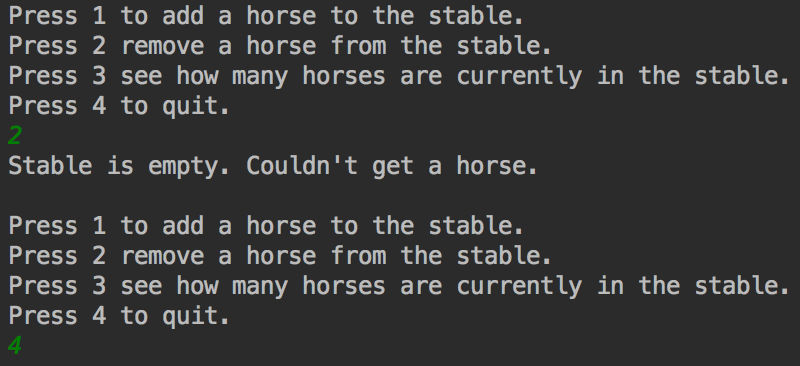
****

*Adding a horse, checking how many there are, removing a horse*

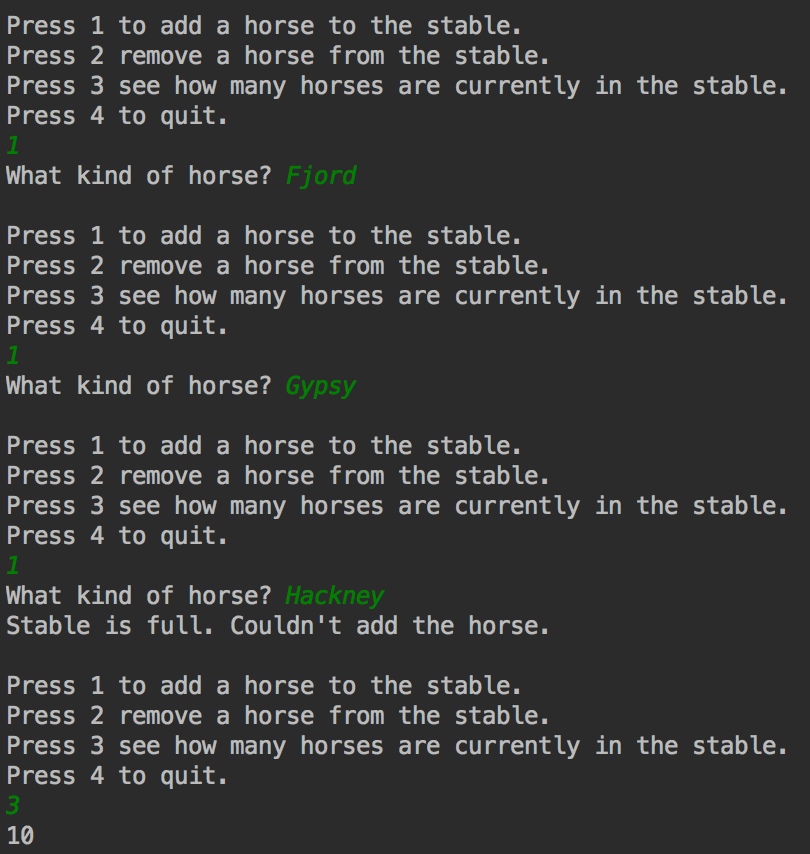
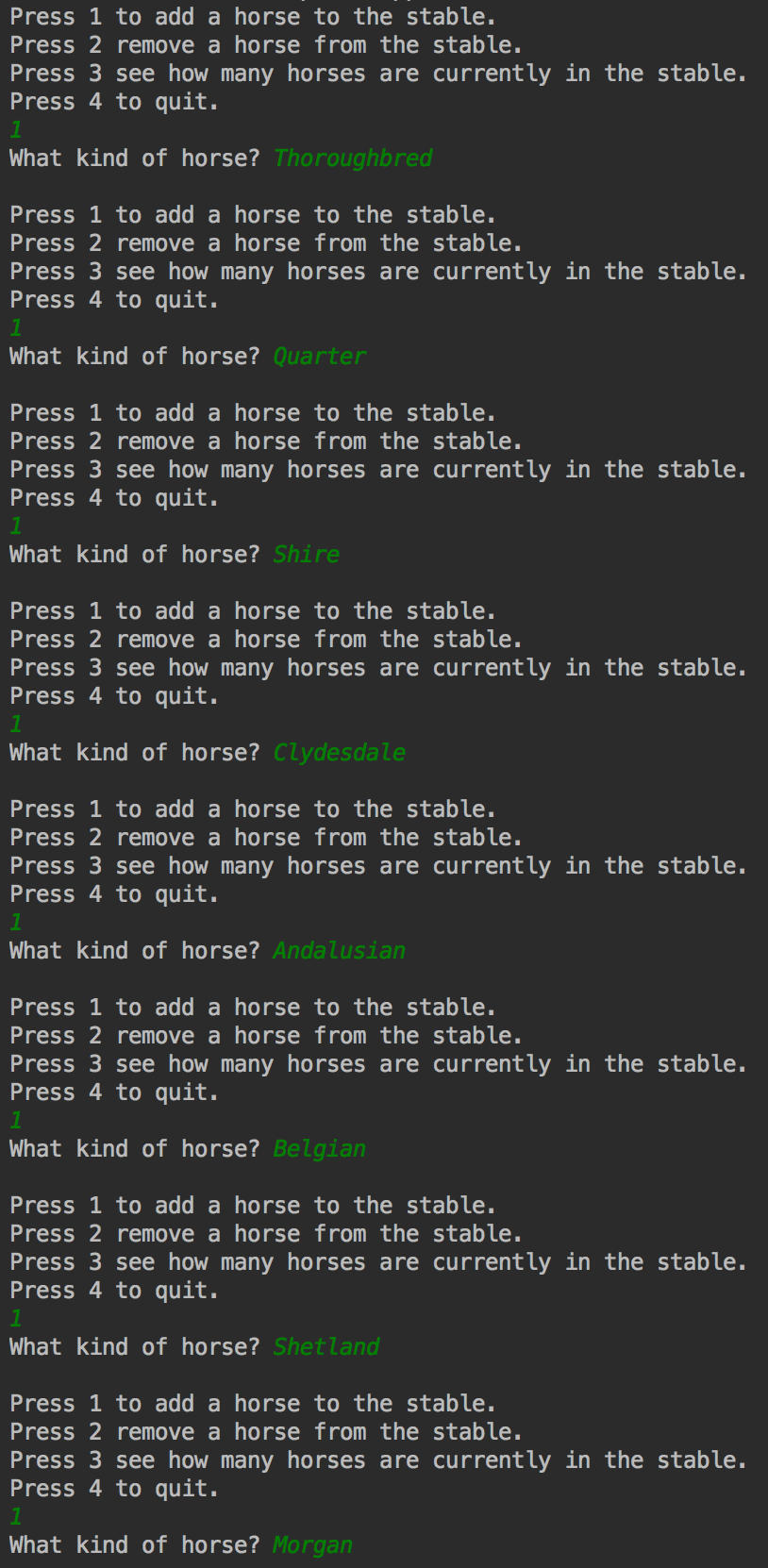


*Adding multiple horses, removing multiple horses*

*Adding multiple horses, checking for a correct number, removing one*

**Task 3**

*Left: Too many horses exception; Top: No horses exception*



In C++, as with many other exception-based languages, the use of exceptions is explicitly encouraged by the language through official exception patterns and try/catch blocks. Using exceptions in C++ allows for many advantages that can be separated into a few categories:

**Readability**

Exception-based code is usually easier to read and marks a clear delineation between normal execution and error handling. This makes exceptions largely invisible until they become necessary, at which point the code to handle them is encapsulated in a catch block and easy to find. When adding future functionality, remembering to add manual error checking is unnecessary and allows for safer extensibility. Finally, allowing the compiler to handle error checking through exceptions means that cleanup (destructors, removing locals, etc.) is handled automatically and with fewer bugs since the programmer does not have to handle the code path.

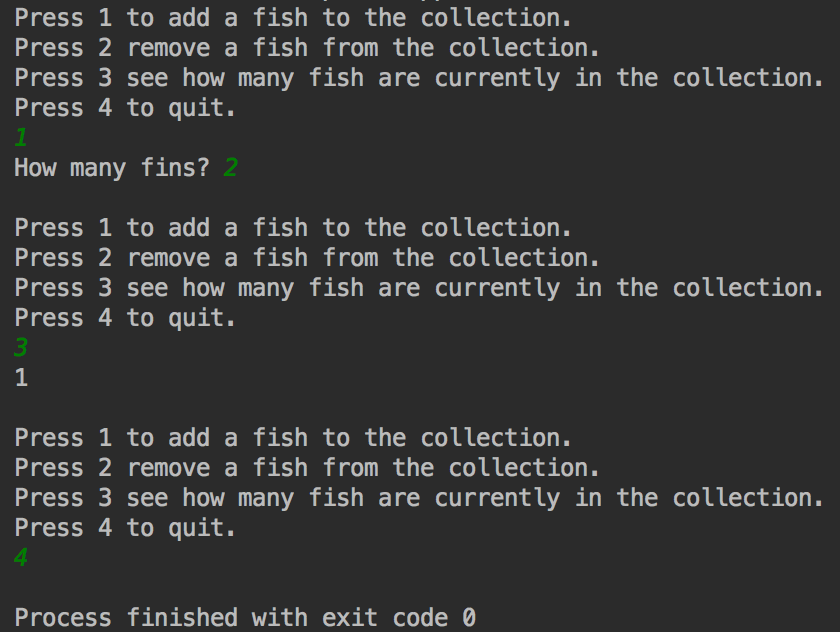
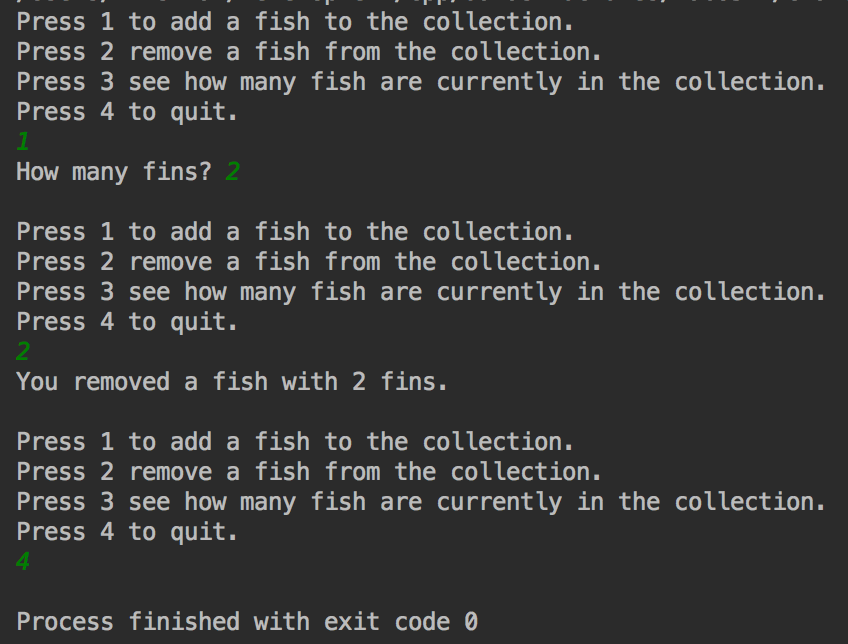
**Difficulty to Avoid**

While you can write a try-catch block that swallows an exception and does nothing, it requires far more effort than ignoring an error code and, due to the structure of the blocks, is easier to spot when debugging and reviewing code. Ignoring an error code is relatively easy because the effort required to do so is far lower. Additionally, a program crashing due to an unhandled exception is usually flagged by most debuggers and can be used as a breakpoint, making forgetting to handle them even harder.

**Direct Access to More Information**

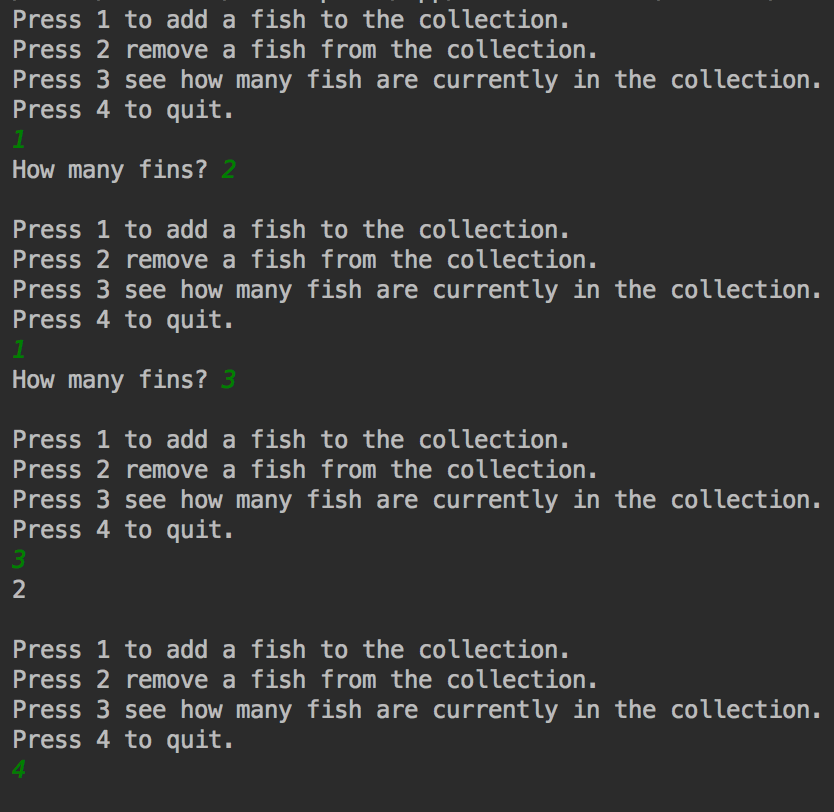
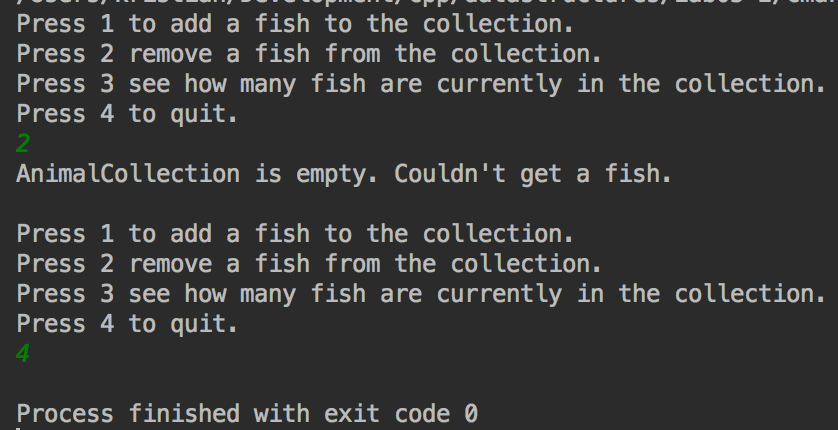
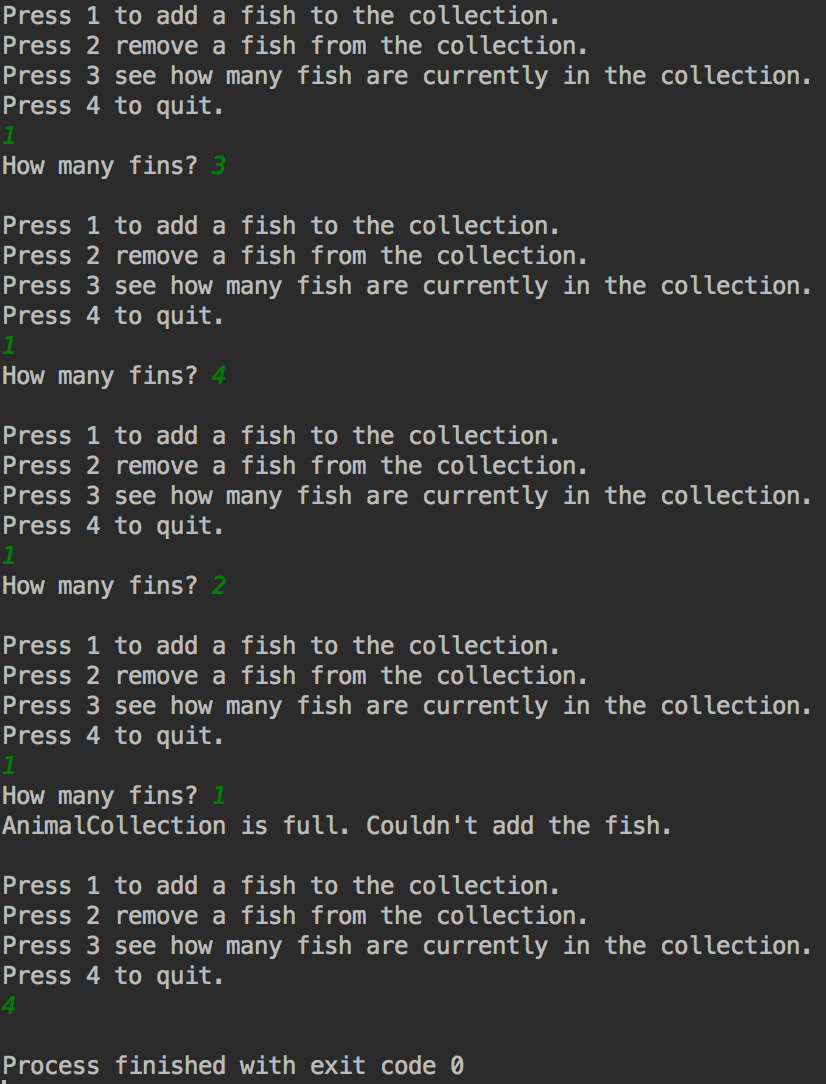
Usually, exceptions are lauded for carrying more information that might be useful to the developer or the programmer. This isn’t necessarily true; a set of error codes can be just as effective in describing the situation to the program. However, this is usually decoupled from the program in a reference book or a table of codes to examine. Custom exception types allow for the information to be directly embedded into the program and examined by the handler, with different types of exceptions automatically handled by the type checker and multiple typed catch blocks. This streamlines the process and makes it more obvious when the program is trying to extract information from an error.

Even with all these benefits, however, there are issues with exceptions. Writing exception-safe code is *hard.* Making sure that the function cleans up correctly by writing correct destructors (and ensuring that they don’t throw exceptions), catches complicated exceptions correctly, is prepared to catch something that may originate far down the call stack, and doesn’t use exceptions for normal code flow are all concepts that must be taught carefully. C++ will not provide any checks to ensure that exception-safe code is written. The most significant issue, however, is that exceptions are extremely expensive. Code written with good error checking will almost always be less resource-intensive than the same concepts executed with exceptions. This means that the developer must be aware of the drawbacks with writing code using them and prevent throwing one as much as possible. While this isn’t necessarily true on systems with large amounts of resources, C++ has an advantage with its performance. Ruining the speed with exceptions hardly seems like a correct strategy.

**Task 4:**

*Adding a fish*

*Removing a fish*



*Checking for multiple fish*

*Attempting to remove a fish from an empty collection – exception handling*

*Adding a fish to a full collection (with max set to* ***3****) – exception handling*