CS105C: Computer Programming C++ Programming Assignment #4

Adrian Trejo Nuñez
atrejo@cs.utexas.edu

Due: 11:59pm October 22, 2018

1 Description

A stream is a special way to represent a sequence of data. The only way to access the elements of a stream is to pull an element from the front. As a result, the $(i + 1)$-th element must be accessed after the i-th element, and usually each element gets processed only once[1]. For example, the TCP packets your computer receives from the network can be thought of as a stream: you grab the packets one at a time and the flow of packets is unidirectional (since TCP has in-order delivery guarantees). Reading files from the disk can also be modeled as reading from a stream of individual bytes or characters, which is why the C++ I/O library is called `iostream`. However, most modern file systems also provide seeking functionality so the I/O operations available today might be more powerful than streams.

At first glance, streams might seem too simple of a tool to to model sequences because they don't offer random access. So why implement a sequence as a stream rather than a vector for instance? A vector lets us access, say, the 10000-th element in constant time, while streams require taking the first 9999 elements and discarding them before we can see the 10000-th element. Even worse, if we don't save the first 9999 elements, there's no way to get them back.

However, the lack of random-access is where the power of streams lies. Since the user can't look into the middle of the stream, there is no point in writing it down yet. Only the first item of the stream needs to be stored and the rest can be generated on-demand. Instead of computing and storing all the elements of the sequence eagerly (like in a vector), a stream let's us store them lazily (as available in functional programming languages like Haskell).

Streams let us represent things like infinite sequences that can't be represented using vectors.

[1] A conveyor belt is a real-world analogy: objects come out of a hole in the wall and you can't "seek" into it and processing an item on the belt is usually a one-time only deal.

You will implement a library of streams in C++. To make our library more usable, the stream class should be polymorphic: a user must be able to create a stream of *any* type. The assignment will help you get used to C++ templates.

2 Directions and Hints

Download the the assignment file `assignment4.tar.gz` to the lab computers (physically or using `scp`). Unpack the tarball in a new directory:

```
> mkdir assignment4
> mv assignment4.tar.gz assignment4
> cd assignment4
> tar -xvf assignment4.tar.gz
```

Take a look at the skeleton code for the various classes. You will be modifying the `Stream` class for the most part.

2.1 Stream Interface

The `Stream` class is implemented as a handle class, meaning it is a wrapper around a pointer to the real implementation of a stream. To make memory management easy, we use `std::shared_ptr` instead of raw pointers so we don't think about allocation. There is some performance hit but it doesn't matter for this assignment.

To represent a stream of any type, the `Stream` class is a template over a value type. The elements pulled from the stream are of the value type. For example, `Stream<int>` represents a stream of integers and `Stream<std::string>` represents a stream of strings.

The interface of the `Stream` class is very simple: it exposes only one member function `next()`. It has two possible outcomes: if the stream is empty then it should return nothing, otherwise it should return the next element of the stream and update the stream. To represent both outcomes, we use the class `Optional<T>` as the return type of the `next()` function.

2.2 `Optional<T>` Class

The `Optional<T>` class is a key part of the API of `Stream` so you should understand how to use it (not necessarily it's implementation)[2].

A value of type `Optional<T>` may or may not hold a value of type `T`. The default constructor creates an `Optional<T>` object with no value in it. An `Optional<T>` object can be initialized or assigned a value of type `T`. You can convert an `Optional<T>` object to boolean to test whether it holds a value or not.

[2]The C++ language committee has voted to add `std::optional<T>` into C++17, which will supersede our custom `Optional<T>` type.

```
Optional<int> opInt;                   // Create an optional with no value
bool hasValue;
hasValue = opInt;                      // hasValue is false
opInt = 3;                             // opInt now holds the value 3
hasValue = opInt;                      // hasValue is true
Optional<std::string> opStr = "abc";  // Create an optional with string "abc"
hasValue = opStr;                      // hasValue is true
```

If an `Optional<T>` object holds a value, you can use the `*` operator to get the value. You should be careful to not extract the value from an empty object.

```
Optional<int> opInt = 10;
if (opInt) {
  int num = *opInt;
  // do something with num
} else {
  // never do *opInt
}
```

In general, the `Optional<T>` class is useful for representing the result of an operation that may fail. In the case of `Stream`, the `next()` function returns an `Optional` object, which is empty if the stream is empty or contains the value that was next on the stream.

The implementation of `Optional<T>` class is provided for you. You can check out `Optional.h` if you're curious, but you don't have to understand the internals.

2.3 An Empty Stream

The simplest stream you can have is a stream that has nothing in it: calling `next()` on the stream always produces an empty value. The implementation of this is very easy and it is provided for you in `EmptyStreamImpl.h`. The `empty()` function defined in `Stream.h` is a helper function that creates an empty stream:

```
auto s = stream::empty<int>();  // Creates an empty stream of integers
for (int i = 0; i < 100; i++)
  assert(!s.next());            // Calling next() always yields an empty value
```

2.4 A Stream Of One Value

The next simplest stream is a singleton stream. The first call `next()` yields a value and all subsequent calls to `next()` yield an empty value. We use the helper function `once()` to create a singleton stream:

```
// Creates a singleton stream that contains 2 only
// Note that we do not need to specify the type of the stream
// It can be deduced from the value passed to once()
```

```
auto s = stream::once(2);
auto opInt = s.next();
// The first element pulled from s0 holds 2
assert(opInt && *opInt == 2);
// Subsequent next() invocation always yield empty value
for (int i = 0; i < 100; i++)
  assert(!s.next());
```

The `once()` function is left for you to implement. It should be very similar to the implementation of `empty()`.

2.5 Stream Concatenation

Now that we have a way to create singleton streams, we can put them together to make more complicated streams. The `chain()` function takes two streams (s_0, s_1) of the same value type and returns a new stream. The `next()` method will first yield elements from s_0 until it is exhausted, and then yield elements from s_1. For example,

```
// Concatenate two singleton streams
// Produce a stream with two elements
auto s = chain(once(2), once(3));
auto first = s.next();
assert(first && *first == 2);
auto second = s.next();
assert(second && *second == 3);

// All elements from both streams are exhausted
assert(!s.next());
```

Using `once()` and `chain()` together allows us to build streams of arbitrary (but finite) length:

```
  // Produce a stream with three elements
  auto s0 = chain(once(1.1), chain(once(1.2), once(1.3)));

  // Produce a stream with size 100
  auto s1 = once(0);
  for (int i = 1; i < 99; ++i)
      s1 = chain(s1, once(i));
```

The `chain()` function is left for you to implement.

2.6 An Infinite Stream (First Attempt)

Up until now, everything we've done with streams can also be achieved with arrays or vectors. We'd like to have streams of infinite length.

The most elegant way of constructing infinite streams uses recursion:

```
Stream<int> repeat() {
  return chain(once(1), repeat());
}
```

This should give an infinite stream that will always give you the value 1. However, this doesn't work since the function call to chain will evaluate the arguments first, before making the call, resulting in infinite recursion.

2.7 Delayed Function Calls

We'd like the stream class to be lazily evaluated, but the C++ language is eagerly evaluated. We want the `repeat()` function call to be evaluated after calling `next()` on the stream, not during construction. The `delay()` helper function takes a callable object as a parameter (the callable object is expected to return a stream) and delays the function call until it is needed. The callable object should to keep track of its own parameters in its state.

The implementation of the `delay()` function is provided for you.

2.8 An Infinite Stream (Second Attempt)

Using `delay()`, we correctly implement the `repeat()` function:

```
Stream<int> repeat() {
  return chain(once(1), delay(repeat));
}
```

Since `repeat()` takes no arguments, it is passed as a function pointer to `delay()`. If we want to delay a function that takes at least one argument, then we need to use lambdas or function objects.

```
Stream<int> counter(int a) {
  return chain(
    once(a),
    delay([a] () { return counter(a + 1); } )
  );
}
```

The `counter()` function takes an integer a and returns a stream that contains an infinite increasing sequence of integers starting from a. The lambda passed to the `delay()` function needs to capture variable a by value. Another interesting example is the function `fib()` that returns a stream of all Fibonacci numbers:

```
Stream<int> fibgen(int a, int b) {
  return chain(
    once(a),
    delay([a, b] () { return fibgen(b, a + b); } )
  );
}

Stream<int> fib() { return fibgen(0, 1); }
```

2.9 Other Helper Functions

To make streams easier to work with, we want other helper functions:

`take()`: Given a stream s and an unsigned number n, produce another stream of size n that contains only the first n elements of s.

```
auto s0 = counter(1);   // s0 = { 1, 2, 3, ... }
auto s1 = take(s0, 2);  // s1 = { 1, 2 }
```

`filter()`: Given a stream s of type T and a callable object f of type $T \to$ `bool`, produce another stream containing only the elements in s for which f returns `true`.

```
auto s0 = counter(1);     // s0 = { 1, 2, 3, ... }

// filter out all odd numbers
auto f = [] (int num) { return num % 2 == 0; };
auto s1 = filter(s0, f);  // s1 = { 2, 4, 6, ... }
```

`map()`: Given a stream s of type S and a callable object f of type $S \to T$, produce another stream of type T which contains the elements of s after applying f to each of them.

```
auto s0 = counter(1);   // s0 = { 1, 2, 3, ... }

// increase each element by 3
auto f = [] (int num) { return num * 3; };
auto s1 = map(s0,  f);  // s1 = { 3, 6, 9, ... }
```

2.10 A Stream Of All Primes

You need to write a function `prime()` that returns a stream consisting of all prime numbers, in ascending order.

The easiest way to do it is to start with the stream `counter(2)` and filter out all numbers that are not prime:

```
bool isPrime(int n) { ... }
Stream<int> prime() {
  return filter(counter(2), isPrime);
}
```

But constructing the stream this way is very inefficient: the function `isPrime()` has to be applied to every number, which tests all potential divisors to see if the number is prime. A more efficient approach uses the Sieve of Eratosthenes.

We start with the stream `counter(2)`. The first element is the first prime. After we remove it, the rest of the stream should not contain anything divisible by 2, and the next element is 3 which is the next prime. To get the next prime, we remove 3 and filter out multiples of 3, and so on.

You must implement the `prime()` function, which should probably look like this:

```
Stream<size_t> sieve(Stream<size_t> s) { ... }
Stream<size_t> prime() {
  return sieve(counter(2));
}
```

The `sieve()` function should be a recursive function that performs the sieving procedure described above. It takes a stream s and uses `filter()` to return another stream containing no multiple of the first element.

To test whether `sieve()` is correct, use the following code to print out the first 20 elements of the stream:

```
auto p = take(prime(), 20);
while (auto elem = p.next())
  std::cout << *elem << '\n';
```

2.11 A Stream Of Hamming Numbers

Your must implement the `hamming()` function that returns a stream of all hamming numbers, in ascending order.

A Hamming number is one that only has prime divisors 2, 3 and 5. An inefficient way of computing the Hamming number stream is:

```
bool isHamming(int n) { ... }
Stream<int> hamming() {
  return filter(counter(2), isHamming);
}
```

But we can do better since the Hamming number stream S has the following properties:

- S begins with 1
- If we multiply each element of S by 2, the resulting elements are also in S
- If we multiply each element of S by 3, the resulting elements are also in S
- If we multiply each element of S by 5, the resulting elements are also in S
- These are all the elements in S

Suppose we have a helper function `mergeUnique()` that takes two increasing streams and combines them into one increasing stream, without repetitions. Then, the Hamming stream S can be implemented by concatenating the singleton stream { `1` } with the `mergeUnique()` of S multiplied by 2, S multiplied by 3, and S multiplied by 5. Multiplying S is easily done with `map()`.

The `mergeUnique()` function will work a lot like merge sort if you have seen it before.

2.12 A Stream Of π

The final task is to write a stream of `double` that converges to π. The value of π can be approximated by truncating the following series:

$$\pi = 4 \times \sum_{i=1}^{\infty} (-1)^{i-1} \times \frac{1}{2 * i - 1} = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...\right)$$

Write a function `pi()` that returns a stream, whose i-th element is the series up to the i-th term. In other words, the first element in the stream is 4.0 (4×1), the second element is 2.66667 ($4 \times (1 - \frac{1}{3})$), the third 3.46667 ($4 \times (1 - \frac{1}{3} + \frac{1}{5})$), and so on.

You probably want to write a helper function `partialSum()` that takes a stream $\{a_0, a_1, a_2, ...\}$ and returns another stream $\{a_0, a_0 + a_1, a_0 + a_1 + a_2, ...\}$.

3 Submission and Grading

- You will submit a single file `assignment4.tar` on Canvas containing all of the files from the assignment, except for `main.cpp`.
- Make sure your final submission compiles without any external dependencies other than the C++ standard libraries. I will compile it using the included Makefile with the command `make`. If your code does not compile, you will receive zero points for this assignment.
- I will automate the grading process by running your program against several test cases (including the examples in this PDF). Make sure the output of your program conforms to the format described in the description section. You should write several test cases yourself and check that your program outputs the correct solution.
- In particular, I will use my own make file that creates several test cases and makes sure your implementation is good. You should use the main file to create your own test cases.
- For the grading script to work, do not change any function signature or remove any existing source files, classes or methods from the skeleton project. You may add extra files, classes, or methods, as long as you modify the Makefile accordingly.

- A small portion of your grade is style, so please make sure you comment your file and you adhere to good coding practices. In particular, you should be consistent throughout your code.
- If you have any questions about the assignment, please ask on Piazza.