Morehead Electronic Journal of Applicable Mathematics
Issue 4 — COSC-2005-01
Copyright ©2005

Implementing Lazy Streams in C++

David Renz and Mike Borowczak[1]

University of Cincinnati, Cincinnati, OH 45219

Abstract

We show that the ability of a lazy language, like Haskell, to allow procedures to lazily generate a stream of tokens can be added to ANSI C++ merely by writing code in a style which uses classes to implement function closures. Coding in this style provides an easy way to handle infinite streams in C++, results in application layer implementations that closely resemble a problem's specification, and can be applied to a wide variety of problems in computer science.

1 Introduction

A *stream* is formally defined as a sequence of tokens x_1, x_2, ..., x_n that are read in increasing order of their indices [GMMO00]. Many applications such as e-mail, internet radio, and news tickers either produce or consume a stream. Tokens in a stream have well-defined types (e.g. integers, Boolean values, strings, user defined types, etc.) that are appropriate for the application making use of the stream. A stream may be thought of as connecting a process which produces tokens, called a *producer*, with a process which consumes those tokens, called a *consumer*. By convention, we call such a connection a *communication link* and only allow tokens to travel in one direction over a link: from producer to the consumer. Observe that it is possible for several communication links to exist from a single producer, that several links can exist to a single consumer, and that a consumer can also be a producer.

A producer may be categorized as eager or lazy, depending on the way it is implemented. An *eager producer* attempts to generate all the tokens in its stream before the consumer sees any one of them. On the other hand, a *lazy producer* generates only those tokens needed to satisfy the demands of a consumer when the consumer makes those demands. In what follows, a stream of tokens generated by a lazy producer will be referred to as a *lazy stream* and a stream of tokens produced by an eager producer will be referred to as an *eager stream*.

[1]This work was done by David Renz and Mike Borowczak as an honors project under the supervision of Dr. John Franco.

Many familiar programming languages only natively support either lazy or eager streams. Some, such as Scheme, have the machinery (namely *lambda* in the case of Scheme) which allows a programmer to build lazy streams. Lazy streams are a standard part of some functional programming languages like Haskell. But, C++ is designed with eager streams in mind at the application layer.

Performing computation with lazy rather than eager streams has at least two benefits. First, a lazy stream is capable of containing an infinite number of tokens. Second, no computational effort is expended until a token is demanded by a consumer. The value of these benefits becomes particularly evident when a consumer extracts a very large, possibly infinite, number of tokens from multiple producers in an order that dependents upon the comparison between first token of the stream generated by each producer. This situation typically results from a recursive specification and requires a way to coordinate the order in which each token is evaluated.

Hamming's problem [Ric97] is a classic illustration of the power of lazy streams, and we use it as our primary example to illustrate their benefits. *Hamming's problem* was originally to find the infinite sequence of integers whose prime factors are all taken from the list [2, 3, 5]. The first ten tokens in this Hamming sequence are 2, 3, 4, 5, 6, 8, 9, 10, 12, and 15. In this paper, we extend Hamming's problem to mean finding the Hamming sequence for any list of prime integers [Feo92]. For example, the first ten tokens in the Hamming sequence for the primes list [3, 5, 11] are 3, 5, 9, 11, 15, 25, 27, 33, 45, and 55. For simplicity, we will assume that any given list of primes is in increasing order.

We can specify a Hamming sequence in terms of the multiplication of a lazy stream with an integer and the merging of two lazy streams of increasing integers. In this example, we will only consider streams of integer tokens, called *integer streams*, so we will write streams as comma separated lists of integers surrounded by brackets (this says nothing about when the tokens are generated; if anything, it just lists all the tokens of a stream). Let `times` be a function that multiplies all tokens of an integer stream by a given integer. That is, `times` takes an integer and a lazy integer stream as arguments and returns a lazy integer stream whose tokens are each the product of the given integer and a token from the input stream. For example, using the notation of Haskell,

```
times (3) ([1, 2, 3, 4])
```

returns [3, 6, 9, 12]. Let `merge` be a function that joins two increasing lazy streams into one increasing lazy stream. That is, `merge` takes two lazy integer streams as arguments and returns a lazy integer stream containing all of the tokens of the two input streams in increasing order. For example,

```
merge ([3, 6, 9, 12]) ([5, 10, 15, 20])
```

returns [3, 5, 6, 9, 10, 12, 15, 20].

The Hamming sequence for any given list of primes may be defined recursively as the concatenation of the first integer in the given list of primes with the `merge` of two other sequences: the first sequence is the `times` of all of the integers in the Hamming sequence of the given list

and first integer in the given primes list and the second sequence contains all of the integers in the Hamming sequence for the given primes list excluding the list's least valued integer. Both a Haskell implementation and a formal recursive specification of a Hamming sequence, as described above, are given in Figure 1.1.

Figure 1.1:

```
hamming [] = []
hamming p = head p : merge (times (head p) (hamming p)) (hamming (tail p))

where

times (a) [] = []
times (a) (b) = (a * (head b) : times (a) (tail b))

merge [] [] = []
merge (a) [] = a
merge [] (b) = b
merge (a) (b) = if ((head a) < (head b)) then ((head a) : merge (tail a) (b))
                                         else ((head b) : merge (a) (tail b))
```

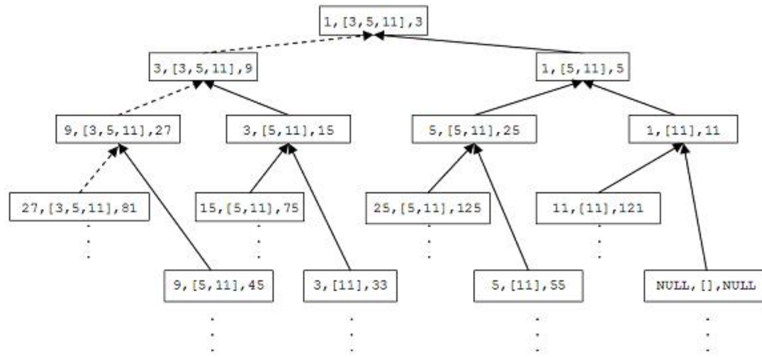
The syntax of the Haskell code in Figure 1.1 is easily interpreted. The Haskell functions `times` and `merge` are defined over more than one line. Each line has an equals sign (=). On the left side of any equals sign is a pattern that corresponds either to the general or special cases of the argument patterns accepted by the function. When the pattern of the accepted arguments matches the pattern specified on the left side of one of the function's specification lines, the function call evaluates to the expression stated on the right of that line's equals sign. For example, if a `hamming` function receives a [] (signifying an empty list) as input, it evaluates to a []; if `merge` receives an empty list and a non-empty list as arguments, it evaluates to the non-empty list (there are two lines for this special case).

Some special symbols and functions are used to denote familiar list operations which are adapted to streams. The function `head` takes one stream argument and returns the first token of that stream. The function `tail` takes one stream argument and returns a stream identical to the given stream minus its first token. The symbol : denotes a binary operation that means concatenate the token on the left side of : with the stream on the right side (that is, insert the left side token into the beginning of the right side stream). The symbols `*`, `<`, `if`, `then`, `else` all have the obvious meaning. Thus, the second line of the specification of `hamming`, namely the line with `hamming p` to the left of the equals, means for non-empty primes list `p`, concatenate the first integer of `p` with the `merge` of the stream obtained by multiplying the first integer of `p` by the integers of the Hamming sequence for `p` and the Hamming sequence for `p` minus the first integer of `p`. In a lazy language, when `hamming p` is invoked, the first token is evaluated (by `head p`) and four lazy streams are created: the first stream's producer is due to the call `hamming p` on the right side of the equals, a second stream's producer is due to the call `hamming` (`tail p`), the producer due to the call to `times` consumes tokens from the first stream, and the producer due to the call to `merge` consumes tokens from the `times` and second streams.

The solution described in Figure 1.1 produces a complex network of increasing integer streams, depicted as nodes in Figure 1.2. Each node consists of a multiplier, a list of prime integers, and a first token. This is enough information to describe an infinite integer stream whose tokens are a Hamming sequence, defined by the node's primes list, all multiplied by the node's multiplier. Each node with a non-empty primes list has two dependencies, called *descendent nodes*. A node with an empty primes list has no descendents. Each dependency is represented in the Figure by a directed arc drawn from a descendent node. The tokens of a node's stream are the merge of the tokens of the streams of its descendents concatenated with the node's first token. The stream of the root node is the Hamming sequence for [3, 5, 11], the stream of the root node's left descendent is 3 times the Hamming sequence for [3, 5, 11], the stream of the root node's right descendent is the Hamming sequence for [5, 11], and so on.

Figure 1.2 directly depicts code relationships. The arcs represent function calls. Each descendent node represents a producer and the node on the other end of an arc represents a consumer.

Figure 1.2:



A C++ function defined by the specification in Figure 1.1 to compute a Hamming sequence cannot produce the same results as the Haskell implementation. In fact, once such a C++ function is invoked, it would never terminate because a C++ function eagerly attempts to completely evaluate the first term it encounters in an expression. The dotted arcs in Figure 1.2 show how an eager C++ implementation of the specification for a Hamming sequence in Figure 1.1 always calls the producer function represented by each node's left descendent node and always neglects the producer represented by the right descendent node. The Haskell implementation works because it looks at both descendents and only evaluates what is needed to generate the next token.

In this paper, we add some of the power of lazy streams to C++ merely by writing code in a certain style. The resources needed to implement lazy streams are already present in ANSI C++ without the aid of libraries like FC++ [MS03]. The function closures needed to implement lazy streams are easily modeled in C++ by enclosing a function and its local environment variables as members of a class object [Luf95]. Coding in this

style provides an easy way to handle infinite streams in C++, results in application layer implementations that closely resemble a problem's specification, and can be applied to a wide variety of problems in computer science.

2 Implementation

In this section, we develop the mechanisms to implement lazy streams in C++ and provide examples of how to apply them. For simplicity, we chose to implement lazy streams that are limited to generating tokens that are integers.

2.1 A Stream Class

The abstract class Stream in Figure 2.1 is inherited by classes that are defined in a consistent style to facilitate the lazy production of a stream of tokens. Classes derived from `Stream` are referred to as *stream classes*, and an instance of a stream class is referred to as a *stream object*. A stream object is a lazy producer.

Figure 2.1:

```
class Stream {
public:
    int head;
    bool isEmpty;
    virtual Stream *tail() = NULL;

    Stream() {isEmpty = false;}
};
```

Inheriting the `Stream` class provides instances of every stream class with certain services. Each stream object has a data member, `head`, which is declared to be an integer and contains the stream's first token. `head`'s value corresponds to the value returned by Haskell's `head` function when given a stream as input. Furthermore, it is `head`'s data type that limits stream objects derived from this `Stream` class to generating tokens that are integers. General stream objects capable of generating tokens of any type are possible if `head`'s data type is changed to `void *`. We will demonstrate the use of such a generalized `Stream` class in the last example of Section 3. Each stream object also has a Boolean data member, `isEmpty`. When set to `true`, `isEmpty` indicates that a stream object is no longer able to produce new tokens and will be referred to as a *null stream*. Since a stream object's constructor will always decide `isEmpty`'s value (which is usually `false` upon creation), initializing `isEmpty` to `false` can considerably shorten a stream class's constructor code. Each stream object also has a member function, `tail()`, which is declared a pure virtual function because it must be explicitly defined for each stream class to produce the remainder of a stream's tokens. Invoking a stream object's `tail()` corresponds to the action performed when Haskell's `tail` function is given a stream as input.

The way a stream class's constructor and `tail()` are implemented differentiates that stream class from other stream classes, but the constructor and `tail()` have the same purpose in all stream implementations. When any new stream object is created, its constructor always eagerly performs some computation to generate its head, makes sure the stream has not become null, and saves any information needed by `tail()` to produce the remainder of the stream's tokens. Invoking `tail()` returns a reference to a new stream object whose constructor eagerly generates the next token. The production of tokens is then suspended until the consumer demands the next token.

It should be noted that, unlike a Haskell stream, a stream object is not fully lazy. In Haskell, a stream's first token is not produced until a consumer demands it. However, a stream object's constructor always eagerly generates its stream's first token when the object is created.

The class hierarchy that results from each stream class inheriting `Stream` allows a stream class's instance to be interchangeable with any other stream class's instance that produces tokens of the same data type. This is possible because declaring `tail()` to be a pure virtual function and casting stream objects to type `Stream` enables C++ to ensure that the correct stream object's `tail()` is always called when a consumer demands the stream's next token. To a consumer, it is not important how a stream class's constructor and `tail()` are implemented as long as the constructor always generates a new value for `head` and invoking `tail()` always returns a new stream object.

2.2 A Successor Stream Class

The `Successor` stream class in Figure 2.2 is a simple example of how a lazy stream may be implemented. A `Successor` stream object is solely a producer of tokens that could, in theory, generate every integer starting from a given integer to infinity in increasing order.

Figure 2.2:

```
class Successor : public Stream{
public:
    Successor( int i ){
        head = i;
    }
    Stream *tail(){ return new Successor( head + 1 ); }
};
```

We will use the function defined in Figure 2.3 as a consumer for all of the examples in Section 2. It consumes up to twenty tokens generated by any given integer stream object. After head is printed to the screen, the next token is demanded by `s->tail()`. This returns a reference to a new stream object, replacing the current stream object referenced by `s`. The generation of tokens is then suspended until the `for` loop's next iteration.

Figure 2.3:

```
void consumer( Stream *s ){
```

```
    for ( int i = 0; ( i < 20 && !s->isEmpty ); i++ ){
        cout << s->head  << endl;
        s = s->tail();
    }
}
```

The above consumer illustrates one drawback to programming in this style: the need for garbage collection. When the consumer replaces s with the stream object's reference returned by `s->tail()`, there is often no longer any reference to the object that s had pointed to, but the operating system is unable to reclaim the object's memory. We did not attempt to provide garbage collection because we are only interested in the concept of programming in this style.

Figure 2.4 shows how a new Successor stream would be created and passed to the consumer function in Figure 2.3, where its first twenty tokens are extracted.

Figure 2.4:

```
Stream *s = new Successor( 1 );
consumer( s );
```

2.3 A Times Stream Class

The `Times` steam class in Figure 2.5 implements what could be considered a stream filter. A `Times` stream object produces a sequence whose tokens are each the product of a token consumed from an input stream object and a given integer multiplier. A `Times` stream object is null if its input stream object is null and performs a task similar to that of the Haskell `times` function defined in Figure 1.1.

Figure 2.5:

```
class Times : public Stream {
    Stream *inStream;
    int multiplier;
public:
    Times( int n, Stream *s ) {
        multiplier = n;
        inStream = s;
 isEmpty = inStream->isEmpty;
        if ( !isEmpty ) head =  multiplier * inStream->head;
    }

    Stream *tail() { return new Times( multiplier, inStream->tail() ); }
};
```

Figure 2.6 creates a new `Times` stream object, `t`, that takes a `Successor` stream object as input and passes `t` to the consumer function from Figure 2.3.

Figure 2.6:

```
Stream *s = new Successor( 1 );
Stream *t = new Times( 5, s );
consumer( t );
```

2.4 A Merge Stream Class

The `Merge` stream class implemented in Figure 2.7 joins two increasing integer streams. It produces an increasing sequence whose tokens are extracted from two input stream objects. A `Merge` stream object is null when both of its input stream objects are null and performs a task similar to the five-line Haskell `merge` function defined in Figure 1.1.

Figure 2.7:

```
class Merge : public Stream {
    Stream *branch1, *branch2;
public:
    Merge ( Stream *a, Stream *b ) {
        if ( a->isEmpty && b->isEmpty ) {
            branch1 = a;
            branch2 = b;
     isEmpty = true;
        } else if ( b->isEmpty ) {
            branch1 = a;
            branch2 = b;
        } else if ( a->isEmpty ) {
            branch1 = b;
            branch2 = a;
        } else if ( a->head < b->head) {
            branch1 = a;
            branch2 = b;
        } else {
            branch1 = b;
            branch2 = a;
        }
        head = branch1-> head;
    }

    Stream *tail() { return new Merge( branch1->tail(), branch2 ); }
};
```

Figure 2.8 creates a `Merge` stream object, `m`, that joins two `Times` stream objects, each of which consume tokens from different `Successor` stream objects, and passes `m` to the consumer.

Figure 2.8:

```
Stream *s1 = new Successor( 1 );
Stream *t1 = new Times( 2, s1 );
Stream *s2 = new Successor( 1 );
Stream *t2 = new Times( 3, s2 );
Stream *m = new Merge( t1, t2);
consumer( m );
```

The diagram in Figure 2.9 illustrates the resulting relationship between the `Successor`, `Times`, and `Merge` stream objects. The solid boxes represent stream objects, and the arrows connecting them are directed from producer stream objects to consumer stream objects. The ovals enclosed in the large dotted boxes represent the first few of the infinite number of tokens that are produced by the `Times` and `Merge` stream objects. The arrows that connect the tokens show the order in which the tokens produced by the `Times` stream objects are put into the sequence produced by `Merge`.

Figure 2.9:

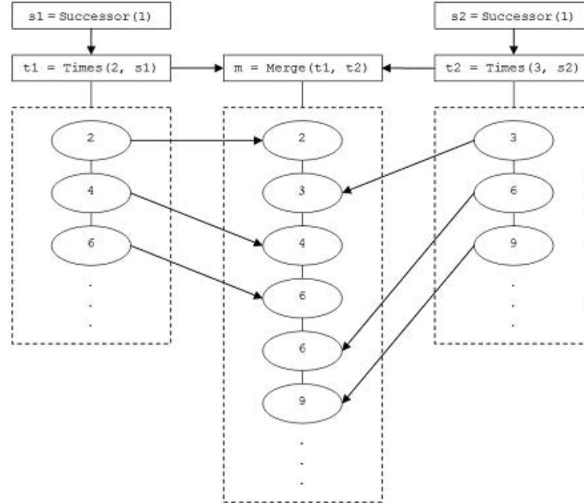


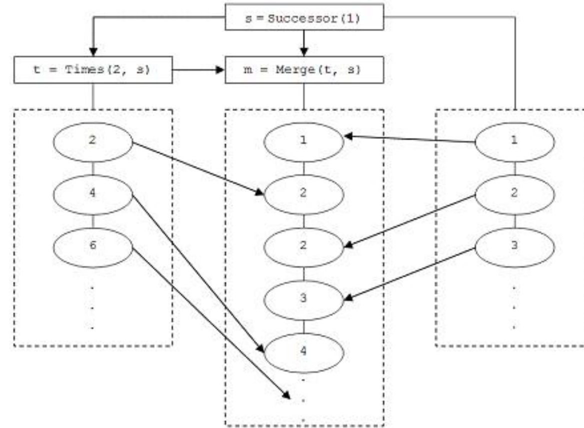
Figure 2.10 creates a single `Successor` stream object that is split into two separate streams by producing tokens for two consumers. Initially, both `Times` and `Merge` demand a token from the same `Successor` stream object. However, two separate, but identical, stream objects are created as the `Successor` stream object's `tail()` is invoked by each consumer.

Figure 2.10:

```
Stream *s = new Successor( 1 );
Stream *t = new Times( 2, s );
Stream *m = new Merge( t, s );
consumer( m );
```

Figure 2.11 shows the relationship between the `Successor`, `Times`, and `Merge` stream objects. As in Figure 2.9, the small solid boxes represent stream objects, and the arrows connecting them are directed from a producer stream object to a consumer stream object. The arrows that connect the ovals show the order in which the tokens extracted from the `Times` and `Successor` stream objects are put into the sequence produced by `Merge`.

Figure 2.11:



2.5 A Hamming Stream Class

To define a stream class whose instances are capable of generating Hamming sequences, it is convenient to represent a list of prime integers as a lazy stream. To do this, we define the `List` stream class in Figure 2.12 to be a bounded stream that holds a list of integers and knows how many items the list contains. A `List` class stream object takes an array of integers and the size of that array as input and produces a lazy stream whose tokens are the elements of the array. A `List` stream object is null when it contains no more items.

Figure 2.12:

```
class List: public Stream {
    int *tokens;
    int size;
public:
    List(int *t, int s ){
      tokens = t;
      size = s;
      if ( size > 0 )
        head = tokens[0];
      else
        isEmpty = true;
    }

    Stream *tail(){ return new List( &tokens[1],(size - 1) ); }
};
```

The `Hamming` stream class, defined in Figure 2.13, implements a stream object that generates a Hamming sequence for any given list of prime integers. This stream object's implementation is similar to Figure 1.1's Haskell specification for generating a Hamming sequence. Like the definition of the Haskell `hamming` function, a `Hamming` stream object is recur-

sively defined to consume tokens from `Merge` and `Times` stream objects that take `Hamming` stream objects as input. Note that the code to the right of the colon in the second line of the Haskell `hamming` function corresponds to the `tail()` function of the `Hamming` stream class. Moreover, notice that in Haskell's `hamming`, a new instance of the `hamming` function with the same primes list must be created for input to `times`. However, the `Hamming` stream object's `tail()` can simply pass the `Times` stream object's constructor the `this` pointer because, as we showed in the second `Merge` stream example, a single stream object may produce multiple independent and identical stream objects.

Figure 2.13:

```
class Hamming : public Stream {
    Stream *p;
public:
    Hamming( Stream *givenPrimes ) {
      p = givenPrimes;
      isEmpty = p->isEmpty;
      if(!isEmpty) head = p->head;
    }

    Stream *tail() {
      return new Merge(new Times(p->head, this),new Hamming(p->tail()));
    }
};
```

Figure 2.14 shows the creation of a new `Hamming` stream object which is passed a `List` stream object. Passing the `Hamming` stream object to the consumer function in Figure 2.3 will result in the generation of the first twenty tokens in the Hamming sequence for the primes list [3, 5, 11].

Figure 2.14:

```
int primes[3]= { 3, 5, 11 };
Stream *h = new Hamming( new List(primes, 3) );
consumer( h );
```

3 Further Examples

This section provides examples that highlight the versatility of using streams in the style we have presented by implementing stream objects as alternatives to more conventional C++ solutions. As examples, we implement stream objects that compute Stirling numbers of the second kind and perform a topological sort.

3.1 Stirling Numbers of the Second Kind

A Stirling number of the second kind, denoted `S(m, n)`, is the number of ways to partition a set of n elements into m nonempty sets [Wei99] and is recursively defined as:

```
S(m, n) = S(m-1, n-1) + m * S(m, n-1)
```
[Dic03]

A `Stirling2` stream object implemented in Figure 3.1 is, in theory, capable of producing all of Stirling numbers of the second kind starting from some initial values of m and n.

Figure 3.1:

```
class Stirling2 :public Stream{
public:
    int m,n;
    int term1, term2;
    Stirling2( int x, int y ) {
        m = x;
        n = y;
        if ( m == n ){
            head = 1;
            m = 0;
            ++n;
        } else if ( m == 1 ){
            head = 1;
        } else if ( m == 0 || n < m ){
            head = 0;
        } else {
            term1 = ( new Stirling2( m-1, n-1 ) )->head;
            term2 = ( new Stirling2( m, n-1 ) )->head;
            head = term1 + m * term2;
        }
    }
    Stream *tail(){ return new Stirling2( m+1, n ); }
};
```

Figure 3.2 defines a consumer function that displays a table of Stirling numbers of the second kind using a `Stirling2` stream object whose initial values of m and n are 1. Sample output produced by the consumer is provided in Figure 7.1 of the Appendix.

Figure 3.2:

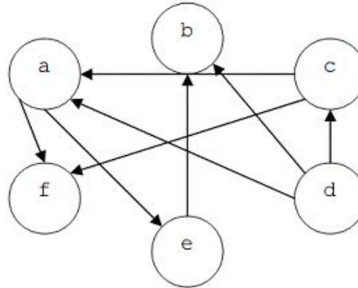
```
void consumer( ) {

    Stream *s = new Stirling2(1,1);
    cout << "n/m";
    for ( int col = 1; col <= 8; col++ ) cout << '\t' << col << " ";
    cout << endl;
    for ( int n = 1; n <= 8; n++ ){
        cout << n << "  \t";
        for ( int m = 1; m <= n; m++ ){
            cout << s->head << '\t';
            s = s->tail();
        }
        cout << endl;
    }
}
```

3.2 Topological Sort

A *topological sort* creates a total ordering from a partial ordering. The directed acyclic graph (DAG) in Figure 3.3 represents a partial ordering. Each directed arc from vertex v_1 to v_2 defines a dependency. A topological sort would order the DAG's vertices such that v_2 must show up later in the total ordering than v_1. One possible result of topologically sorting the

partial ordering represented by this DAG would be to order vertices as follows: d, c, a, e, f, b. This is only one possible solution because more than one correct total ordering for a partial ordering may exist.

Figure 3.3:



The `Topo` stream class implemented in Figure 3.4 performs a topological sort on a given partial ordering of `Vertex` objects. Each `Vertex` object represents a vertex of a DAG and is defined by the class in Figure 7.4 of the Appendix. An array of pointers to `Vertex` objects may be considered a partial ordering because each `Vertex` object also contains an array of pointers to `Vertex` objects, called `depends`, that it is dependent upon. A `Topo` stream object takes an array of pointers to `Vertex` objects and the size of the array as input and lazily produces a sequence whose tokens are pointers to `Vertex` objects. These objects are extracted in an order corresponding to the total ordering of the DAG's vertices. If the input is not a partial ordering (i.e. at least two `Vertex` objects depend upon each other), an error message is displayed and the computation is terminated. A `Topo` stream object becomes null when all of the `Vertex` objects in its given array have been extracted.

The `Topo` stream class must inherit a generalized `Stream` class because the tokens that it produces are pointers to `Vertex` objects. Figure 7.2 provides a `Stream` class in which `head`'s data type is changed to `void *`. This allows instances of stream classes derived from this new `Stream` class to generate tokens that are pointers to objects of any data type.

Figure 3.4:

```
class Topo :public Stream {

  Vertex **vertices;
  int nOfVerts;
public:
  Topo (Vertex **v, int n) {

    vertices = v;
    nOfVerts = n;
    Stream *current;

    for ( int i = 0; i < nOfVerts; i++ ){

      if ( vertices[i]->error ){

        cout << "\nA cycle was found" <<  endl;
        exit(1);

      }
```

```
      if ( vertices[i]->done ) continue;

      vertices[i]->error = true;
      current = new Topo( vertices[i]->depends, vertices[i]->ndepends );
      vertices[i]->error = false;

      if ( current->isEmpty ) current->head = (void *)vertices[i];

      head = current->head;
      ((Vertex*)head)->done = true;
      return;

    }
    isEmpty = true;
  }

  Stream *tail() { return new Topo( vertices, nOfVerts ); }
};
```

The `consumer` function in Figure 3.5 creates a new `Topo` stream object from information in a given formatted file that contains the information needed to represent a DAG, translates this file into an array of pointers to `Vertex` objects, and displays the a total ordering of the represented DAG. A sample input file modeled after Figure 3.3's DAG is provided in Figure 7.3. Figure 7.5 defines functions used by this consumer function to translate the input file into an array of pointers to `Vertex` objects, called `vertices`.

Figure 3.5:

```
void consumer( char * fileName ){

    int nOfVerts = 0;
    fstream fin(fileName, ios::in);

    if (!fin){
        cout << "No such file: " << fileName << endl;
        exit(0);
    }

    nOfVerts = getNumVertices(fileName);
    Vertex **vertices;
    vertices = setUpArray(fileName, nOfVerts);
    setUpDependencies(fileName, nOfVerts, vertices);

    Stream *s = new Topo(vertices, nOfVerts);

    while ( !s->isEmpty ){
        cout << ( (Vertex *)(s->head) )->ident << endl;
        s = s->tail();
    }
}
```

4 Conclusions

We have presented a style of programming in C++ that allows a producer to lazily generate a stream of tokens. Not only does this simple style provide an easy way to handle infinite streams in C++, but it also may enable the implementation of a problem's solution to more closely resemble its specification than a non-stream-based C++ implementation. In the future, it would be interesting to implement the solutions to other classic computer science problems in the style that we have presented.

5 Acknowledgments

We would like to thank Dr. John Franco for the motivation to write this paper and all of his guidance, suggestions, and thorough proofreading.

6 References

[Dic03] R. Dickua. The Math Forum. 2003. "Stirling numbers of the second kind." Accessed April, 12 2003. http://mathforum.org/advanced/robertd/stirling2.html.

[Feo92] J. T. Feo, editor. "A Comparative Study of Parallel Programming Languages: The Salishan Problems (Special Topics in Supercomputing, Vol. 6)." North-Holland, 1992.

[GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. "Clustering Data Streams." In Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS). pp 359 - 366. 2000.

[Luf95] K. Lufer. "A Framework for Higher-Order Functions in C++." In Proceedings of the USENIX Conference on Object Oriented Technologies (COOTS). pp 103 - 116. 1995.

[MS03] B. McNamara and Y. Smaragdakis. August 7, 2003. "FC++: Functional Programming in C++." Accessed August 17, 2003. http://www.cc.gatech.edu/ yannis/fc++/.

[Ric97] M. Richards. "The MCPL Programming Manual and User Guide." University of Cambridge, Computer Laboratory. October 1, 1993.

[Wei99] E. W. Weisstein. Wolfram Research. 1999. "Stirling numbers of the Second Kind." Accessed April 12, 2003. http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html.

7 Appendix

Figure 7.1: Stirling Sample Output:

n/m	1	2	3	4	5	6	7	8
1	1							
2	1	1						
3	1	3	1					
4	1	7	6	1				
5	1	15	25	10	1			
6	1	31	90	65	15	1		
7	1	63	301	350	140	21	1	
8	1	127	966	1701	1050	266	28	1

Figure 7.2: A General Stream Class:

```
class Stream {
public:
    void *head;
    bool isEmpty;
    virtual Stream *tail() = NULL;

    Stream() {isEmpty = false;}
};
```

Figure 7.3: A Sample Input File:

```
e a f c d b -
1 -
3 4 -
1 3 -
4 -
-
0 4 -
```

[Note: Spacing is very important]

Figure 7.4: A Vertex Class:

```
class Vertex {
public:
    Vertex **depends;           // Pointers to dependent objects
    int ndepends;               // Number of dependent objects
    bool error;                 // Used to find a cycle
    bool done;      // Indicates if object has been extracted
    char *ident;                // String identifying this object

    Vertex( char *id) {

        ident = new char[strlen(id)+1];
        strcpy(ident, id);
        ndepends = 0;
```

```
        depends = NULL;
        error = false;
        done = false;
    }

                  // copy constructor
    Vertex( const Vertex& aVertex ){

        ident = new char[strlen(aVertex.ident)+1];
        strcpy(ident, aVertex.ident);
        ndepends = aVertex.ndepends;
        depends = aVertex.depends;
        error = aVertex.error;
        done = aVertex.done;
    }

    // Set the dependencies list - constructed from an input file
    void requires( Vertex **dependencies, int ndep ) {

      depends = dependencies;
      ndepends = ndep;

    }
};
```

Figure 7.5: Functions to Read Input File:

```
// Counts the number of objects in an input file
int getNumVertices( char *fileName ) {
    fstream fin( fileName, ios::in );
    char *token = new char[1024];
    int count = 0;
    while ( fin >> token ){
        if ( token[0] == '-' ) break;
        ++count;
    }
    return count;
}

// Inserts the new Vertex objects into an arry
Vertex **setUpArray( char *fileName, int size ) {
    fstream fin(fileName, ios::in);
    char *token = new char[1024];
    Vertex **vertices = new Vertex *[size];
    for ( int i = 0; i < size; i++ ){
        fin >> token;
        vertices[i] = new Vertex ( token );
    }
    return vertices;
}

// Populates each Vertex object's dependency array
void setUpDependencies( char *fileName, int size, Vertex **vertices ) {
    fstream fin( fileName, ios::in );
    fstream::pos_type mark;
    char *token = new char[1024];
    Vertex **dependents;
    int ndep = 0;
    int index = 0;
```

```
    while ( fin >> token ) {
        if ( token[0] == '-' ) break;
    }
    for ( int i = 0; i < size; i++ ) {
        ndep = 0;
        mark = fin.tellg();  // Saves the file cursors spot
        while( fin >> token ){
            if (token[0] == '-') break;
                ndep++;
        }
        dependents = new Vertex*[ndep];
        fin.seekg( mark, ios::beg );

        for ( int j = 0; j < ndep; j++ ) {
            fin >> token;
            for ( int k = 0; k < size; k++ ) {
                if ( k == atoi( token ) ) {
                    dependents[j] = vertices[k];
                    break;
                }
            }
        }
        vertices[i]->requires( dependents, ndep );
        fin >> token;
    }
}
```