Chapter 3. Infrastructure for Star Writers

Authors: *Joseph T. Buck*
Soonhoi Ha
Edward A. Lee

3.1 Introduction

The Ptolemy kernel provides a number of C++ classes that are fairly generic and often prove useful to star writers. Some of these are essential, such as those that handle errors. Complete documentation of the kernel classes is given in *The Kernel Manual* volume of *The Almagest*. Here, we summarize only the most generic of these classes, i.e., the ones that are generally useful to star programmers. All of the classes described here may be used in stars, provided that the star writer includes the appropriate header files. For instance, the entry

```
ccinclude { "pt_fstream.h" }
```

will permit the star to create instances of the basic stream classes (described below) in the body of functions that are defined in the star. If the user wishes to create such an instance as a `private`, `protected`, or `public` member of the star, then the header file needs to be included in the `.h` file, specified as done in the line

```
hinclude { "pt_fstream.h" }
```

in the Printer star defined on page 2-28.

The source code for most of classes and functions described in this section can be found in `$PTOLEMY/src/kernel`. The source code is the ultimate reference. Moreover, since this directory is automatically searched for include files when a star is dynamically linked, no special effort is required to specify where to find the include files.

3.2 Handling Errors

Uniform handling of errors is provided by the `Error` class. The `Error` class provides four static methods, summarized in table 3-1. From within a star definition, it is not necessary to explicitly include the `Error.h` header file. A typical use of the class is shown below:

```
Error::abortRun(*this,"this message is displayed");
```

The notation "`Error::abortRun`" is the way static methods are invoked in C++ without having a pointer to an instance of the `Error` class. The first argument tells the error class which object is flagging the error; this is strongly recommended. The name of the object will be printed along with the error message. Note that the `abortRun` call does not cause an immediate halt. It simply marks a flag that the scheduler must test for.

The table uses standard C++ notation to indicate how to use the methods. The type of the return value and the type of the arguments is given, together with an explanation of each. When an argument has the annotation "= *something*," then this argument is optional. If it is omitted from the call, then the value used will be *something*.

Error class **#include "Error.h"**

method	description
`static void abortRun (`	*signal a fatal error, and request a halt to the run*
`const NamedObj& obj,`	the object triggering the error
`const char*,`	the error message
`const char* = 0,`	optional additional message to concatenate to the error message
`const char* = 0)`	optional additional message to concatenate to the error message
`static void abortRun`	*signal a fatal error, and request a halt to the run*
`const char*,`	the error message
`const char* = 0,`	optional additional message to concatenate to the error message
`const char* = 0)`	optional additional message to concatenate to the error message
`static void error (...)`	*signal an error, without requesting a halt to the run*
`static void message (...)`	*output a message to the user*
`static void warn (...)`	*generate a warning message*

TABLE 3-1: A summary of the static methods in the `Error` class. Each method has two templates, as shown only for the `abortRun` method. The others are the same.

3.3 I/O Classes

Star programmers often need to communicate with the user. The most flexible way to do this is to build a customized, window-based interface, as described in "Using Tcl/Tk" on page 5-1. Often, however, it is sufficient to plot some data or to just construct strings and output them to files or to the standard output[1]. To do the latter, use the classes `pt_ifstream` and `pt_ofstream`, which are derived from the standard C++ stream classes `ifstream` and `ofstream`, respectively. More sophisticated output can be obtained with the `XGraph` class, the histogram classes, and classes that interface to Tk for generating animated, interactive displays. All of these classes are summarized in this section.

3.3.1 Extended input and output stream classes

The `pt_ofstream` class is used in the `Printer` star on page 2-28. Include the header file `pt_fstream.h`. The `pt_ofstream` constructor is invoked in the `setup` method with

1. Note that when users run pigi, the standard output may appear on a window that is buried. The -console option to pigi helps, in that it creates a specific window for the standard output and other interactions with the user. The standard output is much more useful with ptcl, the textual interpreter.

the call to new. It would not do to invoke it in the constructor for the star, since the fileName state would not have been initialized. Notice that the setup method reclaims the memory allocated in previous runs (or previous invocations of the setup method) before creating a new pt_ofstream object. Notice that we are not using a wrapup method to reclaim the memory, since this method is not invoked if an error occurs during a run.

The classes pt_ifstream and pt_ofstream are only a slight extension of the classes ifstream and ofstream. They add the following features:

- First, certain special file names are recognized as arguments to the constructor or to the open method. These file names are <cin>, <cout>, <cerr>, or <clog> (the angle brackets must be part of the string), then the corresponding standard stream of the same name is used for input (pt_ifstream) or output (pt_ofstream). In addition, C standard I/O fans can specify <stdin>, <stdout>, or <stderr>.
- Second, the Ptolemy expandPathName (see table 3-7 on page 3-8) is applied to the filename before it is opened, permitting it to start with ~user or $VAR.
- Finally, if a failure occurs when the file is opened, Error::abortRun is called with an appropriate error message, including the Unix error condition.

These classes can be used for binary character data as well as ASCII.

3.3.2 Generating graphs using the XGraph class

The XGraph class provides an interface to the pxgraph program, used for plotting data on an X window system display. The pxgraph program and all its options are documented in the *User's Manual*. An example of the output from pxgraph is shown in figure 3-1.The most useful methods of the class are summarized in table 3-2.

Using the XGraph class involves an invocation of the initialize method, some number of invocations of the addPoint method, followed by an invocation of the termi-

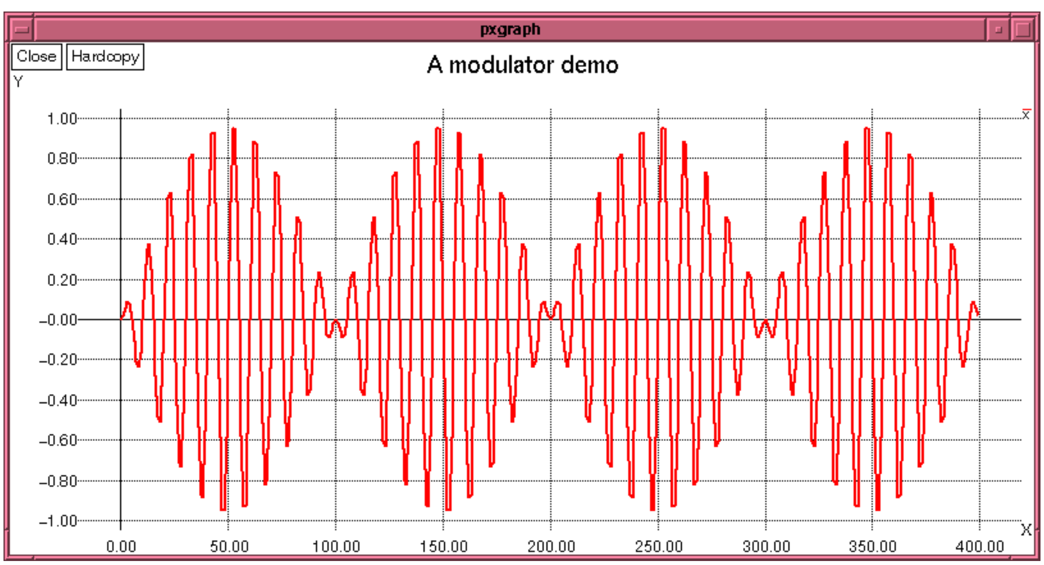

FIGURE 3-1: An example of the output from the pxgraph program, which can be accessed using the XGraph class.

`nate` method. Multiple data sets (currently up to 64) may be plotted together. They will each be given a distinctive color and/or line pattern. Within each data set, it is possible to break the connecting lines between points by calling the `newTrace` method.

XGraph class **#include "Display.h"**

method	description
`void initialize (`	*start a fresh plot*
`Block* parent,`	pointer to the block using the class
`int noGraphs,`	the number of data sets to plot
`const char* options,`	options to pass to the pxgraph program
`const char* title,`	title to put on the graph
`const char* saveFile = 0,`	name of a file to save data to
`int ignore = 0 )`	number of initial points to ignore
`void addPoint (`	*add the next point to the first data set with implicit x position*
`float y )`	the vertical position
`void addPoint (`	*add a single point to the first data set*
`float x,`	the horizontal position of the point to plot
`float y )`	the vertical position of the point to plot
`void addPoint (`	*add a single point to a particular data set*
`int dataSet,`	the number of the data set (starting with 1)
`float x,`	the horizontal position of the point to plot
`float y )`	the vertical position of the point to plot
`void newTrace (`	*start a new trace disconnected from the previous trace*
`int dataSet = 1)`	the data set for the new trace
`void terminate ( )`	*plot the data using the pxgraph program*

TABLE 3-2: A summary of the most useful methods of the `XGraph` class, which provides a simple interface to the `pxgraph` program, used for plotting data.

3.3.3 Classes for displaying animated bar graphs

The `BarGraph` class creates a Tk window that displays a bar graph that can be modified dynamically, while a simulation runs. An example with 12 data sets and 8 bars per data set is shown in figure 3-2. The most useful methods of the class are summarized in table 3-3. This class is directly usable only by stars linked into a `pigi` process, not to stars linked into the interpreter, `ptcl`. The reason for this is that `ptcl` does not have the Tk code linked into it. Correspondingly, the class definition source code is in `$PTOLEMY/src/pigilib`, rather than the more usual `$PTOLEMY/src/kernel`.

BarGraph class **#include "BarGraph.h"**

method	description
`int setup (`	*start a fresh plot; return FALSE if setup fails*
`Block* parent,`	pointer to the block using the class
`char* desc,`	label for the bar graph
`int numInputs,`	the number of data sets to plot
`int numBars`	the number of bars per data set to show at once
`double top,`	the numerical value that will produce the highest bar
`double bottom,`	the numerical value that will produce the lowest bar
`char* geometry,`	the starting position for the window (e.g. “+0+0” or “-0-0”)
`double width,`	the starting width of the window (in cm)
`double height )`	the starting height of the window (in cm)
`int update (`	*modify or add a bar; return FALSE if it fails*
`int dataSet,`	the number of the data set (starting with 0)
`int bar,`	the horizontal position of the point to plot
`double y )`	the requested height of the bar

TABLE 3-3: A summary of the most useful methods of the `BarGraph` class, which creates animated bar graph charts in a window, and is available to stars running under `pigi`.

3.3.4 Collecting statistics using the histogram classes

The `Histogram` class constructs a histogram of data supplied. The `XHistogram`

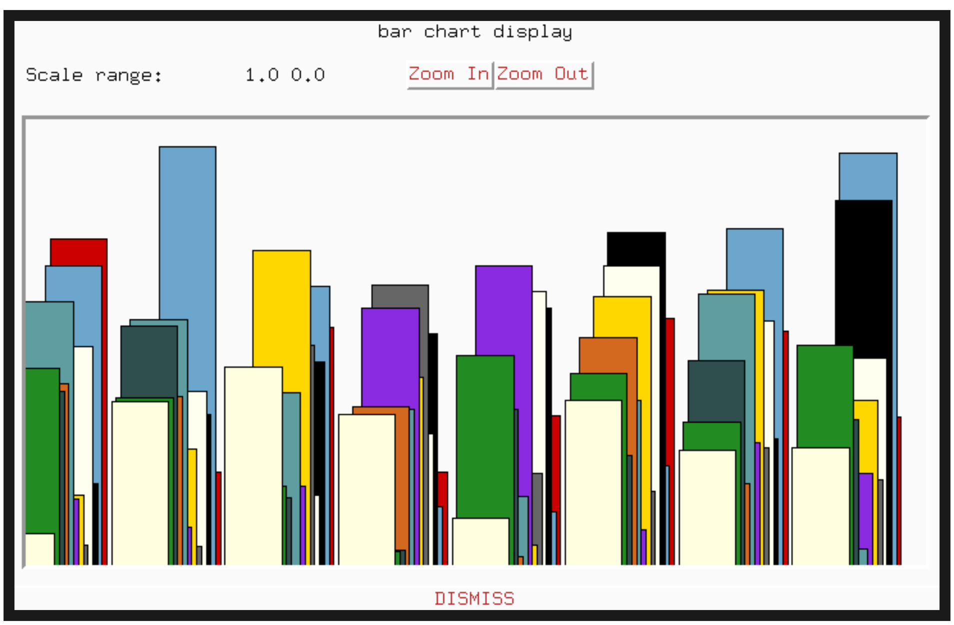

FIGURE 3-2: An example of an animated bar graph created using the `BarGraph` class. This class uses Tk, so it is available under `pigi`, but not under `ptcl`.

class also constructs a histogram, but then plots it using the `pxgraph` program. An example of such a plot is shown in figure 3-3. The most useful methods of both classes are summarized in tables 3-4 and 3-5.

The `Histogram` class counts the number of occurrences of data values that fall within each of a number of bins. Each bin represents a range of numbers. All bins have the same width, and the center of each bin will be an integer multiple of this width. Bin number 0 is always that with the smallest center. Bins are added if new data arrives that does not fit within any of the existing bins. The `getData` method is used to read out the contents of a bin. If you start with bin number 0, and proceed until `getData` returns `FALSE`, you will have read all the bins.

Histogram class **#include "Histogram.h"**

	description
`Histogram (`	*constructor*
`double width = 1.0,`	the width of each bin; bins are centered at integer multiples of this
`int maxBins = 1000 )`	since bins are added as needed, it is wise to limit their number
`void add (`	*add to the count of the bin within which the given data falls*
`double x )`	a data point for the histogram
`int numCounts ( )`	*return the number of data values used so far in the histogram*
`double mean ( )`	*return the average value of all observed data so far*
`double variance ( )`	*return the variance of the observed data so far*
`int getData (`	*get the count for a given bin; return FALSE if the bin is out of range*
`int binno,`	starting at 0, the bin number
`int& count,`	place to store the count for the given bin
`double& binCenter )`	place to store the center of the given bin

TABLE 3-4: A summary of the most useful methods of the `Histogram` class, which creates histogram charts in a window, and is available to stars running under `pigi`.

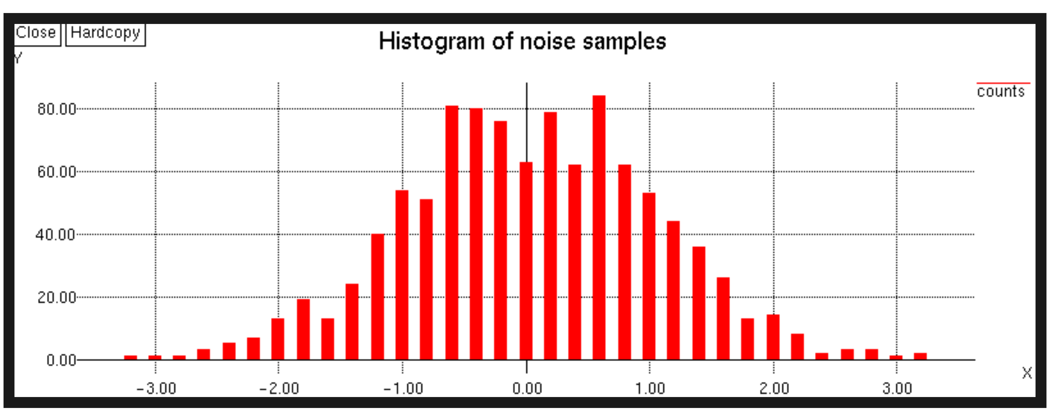

FIGURE 3-3: An example of a histogram generated using the `XHistogram` class.

XHistogram class **#include "Histogram.h"**

method	description
`void initialize (`	*start a fresh histogram*
`Block* parent,`	pointer to the block using the class
`double binWidth,`	the width of each bin; bins are centered at integer multiples of this
`const char* options,`	options to pass to the pxgraph program, in addition to -bar -nl -brw
`const char* title,`	title to put on the histogram
`const char* saveFile,`	name of a file to save data to (or 0 if none)
`int maxBins = 1000 )`	since bins are added as needed, it is wise to limit their number
`void addPoint (`	*add to the count of the bin within which the given data falls*
`double y)`	a data point for the histogram
`int numCounts ( )`	*return the number of data values used so far in the histogram*
`double mean ( )`	*return the average value of all observed data so far*
`double variance ( )`	*return the variance of the observed data so far*
`void terminate ( )`	*plot the histogram using the pxgraph program*

TABLE 3-5: A class for displaying histograms.

3.4 String Functions and Classes

The Ptolemy kernel defines some ordinary functions (not classes) plus two classes that are useful for building and manipulating strings. The non-class string functions are summarized in table 3-6.. These include functions for copying strings, adding strings to a system-

ordinary functions for strings **#include "miscFuncs.h"**

method	description
`char* savestring ( const char* text)`	*create a new copy of the given text and return a pointer to it; the caller must eventually delete the string.*
`const char* hashstring ( const char* text)`	*save a copy of the text in a system-wide hash table, if it isn't already there, and return a pointer to the entry.*
`char* tempFileName ( )`	*return a new, unique temporary file name; the caller must eventually delete the string.*
`const char* expandPathName ( const char* filename)`	*return an expanded version of the filename argument, which may start with "~", "~user", or "$var"; the expanded result is in static storage, and will be overwritten by the next call.*

TABLE 3-6: Non-class (ordinary) functions available in the Ptolemy kernel for string manipulation

wide hash table, creating temporary file names. The non-class pathname functions are summarized in table 3-7. These functions are for expanding file names that might begin with a reference to a user's home directory (`"~username"`) or an shell environment variable (`"$VARIABLE"`). Also provided is a function for verifying that an external program to be invoked is available, and a function for searching the user's path.

ordinary functions for path search **#include "paths.h"**

method	description
`int progNotFound (`	*flag an error and return TRUE if a program is not found*
`const char* program,`	the name of the program to find in the user's path
`const char* extramsg = 0)`	message to add to error message if the program isn't found
`const char* pathSearch (`	*find a file in a Unix-style path, returning the directory name*
`const char* file,`	file name to find in the path
`const char* path = 0)`	if non-zero, the path to use instead of the user's path

TABLE 3-7: Non-class (ordinary) functions available in the Ptolemy kernel for certain pathname manipulations.

Two classes are provided for manipulating strings, `InfString`, and `StringList,` these classes are summarized in figure 3-8.

StringList class **#include "StringList.h"**

method	description
`StringList`	*constructors can take any of the following possible arguments*
`none`	return an empty StringList
`const StringList& s`	copy s and return a new, identical StringList
`char c`	return a StringList with one string of one character
`const char* string`	copy the string and makes a one element StringList containing it

`int i`
`double x`
`unsigned u`
	create an ASCII representation of the number and return a one element StringList with that number as the element

`StringList& operator = arg`	*assignment takes the same types of arguments as the constructors, except "none"*
`StringList& operator << arg`	*add one or more elements to a StringList; this takes the same types of arguments as the constructors, except "none"*
`operator const char*`	*join all elements together and return as a single string;*
`void initialize ( )`	*delete all elements, making the StringList empty*
`int length ( )`	*return the length in characters (sum of the lengths of the elements)*
`int numPieces ( )`	*return the number of elements*
`const char* head ( )`	*return the first element*
`char* newCopy ( )`	*return the concatenated elements in a single newly allocated string; the caller must free the memory allocated.*

InfString class **#include "InfString.h"**

method	description
all `StringList` methods	see above
`operator char*`	*join all elements together and return as a single string;*

TABLE 3-8: A summary of the most useful methods of the `StringList` and `InfString` classes. The `InfString` class inherits all of the methods from `StringList`, adding only the cast to `char*`.

Although these two classes are almost identical in design, their recommended uses are quite different. The first is designed for building up strings without having to be concerned about the ultimate size of the string. New characters can be appended to the string at any time, and memory will be allocated to accommodate them. When you are ready to use the string, perhaps by passing it to a function that expects the standard character array representation of the string, then simply cast the object to `char*`.

In fact, `InfString` is publicly derived from `StringList`, adding only the cast to `char*`. `StringList` is implemented as a list of strings, where the size of the list is not bounded ahead of time. `StringList` is recommended for applications where the list structure is to be preserved. The cast to `char*` in `InfString` destroys the list structure, consolidating all its strings into one contiguous string.

The most useful methods for both classes are summarized in table . Since `InfString` differs by only one operator, we show only that one operator.

A word of warning is in order. If a function or expression returns a `StringList` or `InfString`, and that value is not assigned to a `StringList` or `InfString` variable or reference, and the `(const char*)` or `(char*)` cast is used, it is possible (likely under g++) that the `StringList` or `InfString` temporary will be destroyed too soon, leaving the `const char*` or `char*` pointer pointing to garbage. The solution is to assign the returned value to a local `StringList` or `InfString` before performing the cast. Suppose, for example, that the function `foo` returns an `InfString`. Further, suppose the function `bar` takes a `char*` argument. Then the following code will fail:

```
bar(foo());
```

(Note that the cast to `char*` is implicit). The following code will succeed:

```
InfString x = foo();
bar(x);
```

3.5 Iterators

The `StringList` class is one of several list classes in the Ptolemy kernel. A basic operation on list classes is to sequentially access their members one at a time. This is accomplished using an iterator class, companion to the list class. For the `StringList` class, the iterator is called `StringListIter`. Its methods are summarized in table 3-9. An example program frag-

StringListIter class **#include "StringList.h"**

method	description
`StringList (`	*constructor*
`StringList& list )`	the list over which the iterator will iterate
`const char* next ( )`	*return the next string on the list, or 0 if there are no more*
`const char* operator ++ ( )`	*a synonym for "next"*
`void reset ( )`	*reset the iterator to start at the head again*

TABLE 3-9: An example of an iterator class, used to access the elements of a list class.

ment using this is given below:

```
StringListIter item(myList);
const char* string;
```

```
while ((string = item++) != 0) cout << string << "\n";
```

In this fragment, `myList` is assumed to be a `StringList` previously set up.

3.6 List Classes

The `StringList` class is privately derived from the `SequentialList` class, an extremely useful class used throughout Ptolemy. This class implements a linked list with a running count of the number of elements. It uses the generic pointer technique, with

```
typedef void* Pointer
```

Thus, items in a sequential list can be pointers to any object, with a generic pointer used to access the object. In derived classes, like `StringList`, this generic pointer is converted to a specific type of pointer, like `const char*`. The methods are summarized in table 3-10.

An important point to keep in mind when using a `SequentialList` is that its destructor does not delete the elements in the list. It would not be possible to do so, since it has only a generic pointer. Also, note that random access (by element number, or any other method) can be very inefficient, since it would require sequentially chaining down the list.

`SequentialList` has an iterator class called `ListIter`. The `++` operator (or `next` member function) returns a `Pointer`.

In table 3-11 are two classes privately derived from `SequentialList`, `Queue` and `Stack`. The first of these can implement either a first-in, first-out (FIFO) queue, or a last-in,

SequentialList class **#include "DataStruct.h"**

method	description
`void append (Pointer p)`	*add the element p to the end of the list*
`Pointer elem ( int n )`	*return the n-th element on the list (zero if there are fewer than n)*
`int empty ( )`	*return 1 if empty, 0 if not*
`Pointer getAndRemove ( )`	*return and remove the first element on the list (return zero if empty)*
`Pointer getTailAndRemove ( )`	*return and remove the last element on the list (return zero if empty)*
`Pointer head ( )`	*return the first element on the list (zero if empty)*
`void initialize ( )`	*remove all elements from the list*
`int member (Pointer p)`	*return 1 if the list has a pointer equal to p, 0 if not*
`void prepend (Pointer p)`	*add the element p to the beginning of the list*
`int remove (Pointer p)`	*if the list has a pointer equal to p, remove it, and return 1; 0 if not*
`int size ( )`	*return the number of elements on the list*
`Pointer tail ( )`	*return the last element on the list (zero if empty)*

TABLE 3-10: The most useful basic list structure defined in the Ptolemy kernel.

first-out (LIFO) queue. The second implements a stack, which is also a LIFO queue.

Queue class **#include "DataStruct.h"**

method	description
`Pointer getHead ( )`	*return and remove the first element on the list (return zero if empty)*
`Pointer getTail ( )`	*return and remove the last element on the list (return zero if empty)*
`void initialize ( )`	*remove all elements from the list*
`void putHead (Pointer p)`	*add the element p to the beginning of the list*
`void putTail (Pointer p)`	*add the element p to the end of the list*
`int size ( )`	*return the number of elements on the list*

Stack class **#include "DataStruct.h"**

method	description
`Pointer accessTop ( )`	*return the top of the stack without removing it (return zero if empty)*
`void initialize ( )`	*remove all elements from the list*
`Pointer popTop ( )`	*return and remove the top element from the stack (zero if empty)*
`void pushBottom (Pointer p)`	*add the element p to the bottom of the stack*
`void pushTop (Pointer p)`	*add the element p to the top of the stack*
`int size ( )`	*return the number of elements on the list*

TABLE 3-11: Two classes derived from SequentialList.

3.7 Hash Tables

Hash tables are lists that are indexed by an ASCII string. A “hashing function” is computed from the string to make random accesses reasonably efficient; they are much more efficient, for example, than a linear search over a `SequentialList`. Two such classes are provided in the Ptolemy kernel. The first, `HashTable`, is generic, in that the table entries are of type `Pointer`, and thus can point to any user-defined data structure. The second, `TextTable`, is more specialized; the entries are strings. It is derived from `HashTable`.

The `HashTable` class is summarized in table 3-12 and `TextTable` class is summarized in table 3-13. Only the most useful (and easily used) methods are described. You may want to refer to the source code for more information. The `HashTable` class has a standard iterator called `HashTableIter`, where the `next` method and ++ operator return a pointer to class `HashEntry`. This class has a `const char* key()` method that returns the key for the entry, and a `Pointer value()` method that returns a pointer to the entry. `TextTable` has an iterator called `TextTableIter`, where the `next` method and ++ operator return type `const char*`.

HashTable class **#include "HashTable.h"**

method	description
`void clear ( )`	*empty the table*
`virtual void cleanup ( Pointer p)`	*does nothing; in derived classes, this might deallocate memory*
`int hasKey ( const char* key )`	*return 1 if the given key is in the table, 0 otherwise*
`void insert ( const char* key, Pointer data )`	*insert an entry; any previous entry with the same key is replaced, and the cleanup method is called so that in derived classes, its memory can be deallocated.*
`Pointer lookup ( const char* key)`	*lookup an entry; in a derived class, this could be overloaded to return a pointer of a more specific type.*
`int remove ( const char* key)`	*remove the entry with the given key from the table; note that the object pointed to by the entry is not deallocated.*
`int size ( )`	*return the number of entries in the hash table*

TABLE 3-12: A summary of the most useful methods of the HashTable class

Sophisticated users will often want to derive new classes from `HashTable`. The reason is that the methods that look up data in the table can be defined to return pointers of the appropriate type. Moreover, the deallocation of memory when an entry is deleted or the table itself is deleted can be automated. `TextTable` is a good example of such a derived class. This is not possible with the generic `HashTable` class, because the `Pointer` type does not give enough information to know what destructor to invoke. Thus, when using the generic `HashTable` class, the user should explicitly delete the objects pointed to by the `Pointer` if they were dynamically created and are no longer needed. A detailed example that directly uses the `HashTable` class, without defining a derived class, is given in the next section. In that example, the `Pointer` entries point to stars in a universe, so they should not be deleted when the entries in the table are deleted. Their memory will be deallocated when the universe is deleted.

TextTable class **#include "HashTable.h"**

method	description
`void clear ( )`	*empty the table*
`void cleanup ( Pointer p)`	*deallocate the string pointed to by p*
`int hasKey ( const char* key )`	*return 1 if the given key is in the table, 0 otherwise*
`void insert ( const char* key, const char* string )`	*create an entry containing a copy of string; any previous entry with the same key is replaced, and the cleanup method is called to deallocate its memory.*
`const char* lookup ( const char* key)`	lookup an entry with the given key; return 0 if there is no such entry.
`int remove ( const char* key)`	remove the entry with the given key from the table and deallocated its memory.
int **size** ( )	return the number of entries in the hash table

TABLE 3-13: A summary of the most useful methods of the `HashTable` and `TextTabl` classes.

In some future version, `HashTable` might be reimplemented using templates.

3.8 Sharing Data Structures Across Multiple Stars

It is sometimes desirable to have a set of stars that share and manipulate a common data structure[1]. A simple way to accomplish this is to define a star that contains a static member. Suppose, for example, you wish to define a class of stars that create a shared list of pointers, one to each instance of this type of star. Thus, every star of this type would be able to access every other star of this type. Consider the following implementation:

```
defstar {
      name { Share }
      domain { SDF }
      desc { A star with a shared data structure }
      hinclude { "DataStruct.h" }
      private {
          static SequentialList starList;
      }
      output {
          name { howmany }
          type { int }
      }
      code {
```

1. See the SDFWriteVar and SDFReadVar stars for a similar implementation.

```
            SequentialList SDFShare::starList;
        }
        begin {
            starList.append(this);
        }
        go {
            howmany%0 << starList.size();
        }
        wrapup {
            starList.initialize();
        }
}
```

This star has a static private member of type `SequentialList` with name `starList`. The "`static`" in C++ ensures that there will be no more than one instance of the `SequentialList` object. That instance will be accessible to every instance of the star, but not to any other object (because the member is private). That one instance is actually declared by the lines:

```
code {
    SequentialList SDFShare::starList;
}
```

The declaration will get put into the file `SDFShare.cc` by the preprocessor. Notice that the class name of the star is `SDFShare` not just `Share`. The `begin` method simply adds to the sequential list a pointer to the star that invoked the `begin` method (`this`). Note that you should use the `begin` method here rather than the `setup` method because the `begin` method is always invoked exactly once, while the `setup` method might be invoked more than once when the simulation starts up. The `go` method sends to the output (named `howmany`) the size of the list. This will be equal to the number of stars of this type in the universe.

The `wrapup` method has the only tricky part of this code. It reinitializes the `SequentialList` so that subsequent runs do not just simply add to a list created by previous runs. However, note that the wrapup method will not be invoked if an error occurs during the run. `Pigi` ensures correct operation nonetheless by deleting all instances of the stars and recreating them if an error occurred on the previous run. Thus, between invocations of the `begin` method, either the `wrapup` method or the constructor for the star (and all its members) will be invoked. The constructor for `SequentialList` also initializes the list, so the list is always initialized before the first `begin` method is called.

The above approach is somewhat limited. You may want more than one type of star to share a data structure. In this case, you should create a common base class for all the stars that will share the data structure. The shared data structure should be a protected member, rather than a private member, so that it is accessible to derived stars.

Alternatively, you might want arbitrary subsets of stars to share distinct data structures, one for each subset. This can be accomplished by defining a static list that is indexed by a string, and using a parameter in the star to identify to which subset it belongs. An efficient data structure to use for this is the `HashTable`. So for example, suppose we wanted to modify the above star to create lists of stars with common values of a parameter "mySubset", and to output the number of stars in their subset. The above code becomes:

```
defstar {
```

```
        name { BetterShare }
        domain { SDF }
        desc { A star with a shared data structure }
        hinclude { "DataStruct.h" }
        hinclude { "HashTable.h" }
        output {
            name { howmany }
            type { int }
        }
        state {
            name { mySubset }
            default { "subset A" }
            type { string }
        }
        private {
            static HashTable listOfLists;
            SequentialList* myList;
        }
        code {
            HashTable SDFBetterShare::listOfLists;
        }
        begin {
            if (listOfLists.hasKey((char*)mySubset)) {
              myList = listOfLists.lookup((char*)mySubset);
            } else {
              myList = new SequentialList;
              listOfLists.insert((char*)mySubset,myList);
            }
            myList->append(this);
        }
        go {
            howmany%0 << myList->size();
        }
        wrapup {
            if (listOfLists.hasKey((char*)mySubset)) {
              listOfLists.remove((char*)mySubset);
              delete myList;
            }
        }
}
```

In addition to the static private member `listOfLists`, we also have a pointer `myList` to a `SequentialList`. The `begin` method is a bit more complicated now. It first checks to see whether an entry in the hash table has already been created with a key equal to the value of the state “mySubset”. If it has, then the `SequentialList` pointer `myList` is set equal to the value of that entry. If it has not, then a new `SequentialList` is allocated and inserted into the hash table with the appropriate key. The last action is simply to insert a pointer to the star instance into `myList`.

The `go` method is similar to before.

The `wrapup` method is slightly more complicated, because it needs to free the memory allocated when the new `SequentialList` was allocated. However, it should free that memory only once, and there may be several star instances pointing to it. Thus, it first checks the hash table to see whether there exists an entry with key equal to mySubset. If there does, then it removes that entry and deletes the SequentialList myList.

3.9 Using Random Numbers

Ptolemy uses the Gnu library routines for the random number generation. Refer to Volume II of the Art of Computer Programming by Knuth for details about the method. There are built-in classes for some popular distributions: uniform, exponential, geometric, discrete uniform, normal, log-normal, and so on. These classes use a common basic random number generation routine which is realized in the ACG class. Here are some examples of using random numbers.

The first example is the part of the DE Poisson star. See the DE chapter for details on how to write DE stars.

```
hinclude { <NegExp.h> }
ccinclude { <ACG.h> }
protected {
      NegativeExpntl *random;
}
// declare the static random-number generator in the .cc file
code {
      extern ACG* gen;
}
constructor {
      random = NULL;
}
destructor {
      if(random) delete random;
}
setup {
      if(random) delete random;
      random = new NegativeExpntl(double(meanTime),gen);
      DERepeatStar :: setup();
}
go {
       ......
      // Generate an exponential random variable.
      double p = (*random)();
       ......
}
```

The built-in class for an exponentially distributed random numbers is NegativeExpntl.

The Ptolemy kernel provides a single object to generate a stream of random numbers; the global variable gen (a poor choice of name, perhaps) is a pointer to it. We create an instance of the NegativeExpntl class in the setup method, not in the constructor since Ptolemy allows you to change the seed of the random number generator. When the user changes the seed of the random number generator, the object pointed to by gen is deleted and re-created, so all objects such as the one pointed to by random in this star become invalid.

Finally, we can read a random number of the specific type by calling operator `()` of the `NegativeExpnl` class.

This example uses a uniformly distributed random number.

```
hinclude { <Uniform.h> }
ccinclude { <ACG.h> }
protected {
      Uniform *random;
}
// declare the extern random-number generator in the .cc file
code {
      extern ACG* gen;
}
constructor {
      random = NULL;
}
destructor {
      if(random) delete random;
}
setup {
      if(random) delete random;
      random = new Uniform(0,double(output.numberPorts()),gen);
}
go {
      ......
      double p = (*random)();
      ......
}
```

You may notice that the two examples above are very similar in nature. You can get another kind of distribution for the random numbers, by including the appropriate library file in the `.h` file and by creating the instance with the right parameters in the `setup` method.