Chapter 1. Basic concepts, classes, and facilities

Authors: *The Ptolemy Team*

This section describes some basic classes and low-level concepts that are used throughout Ptolemy. There are a number of iterator classes, all with the same interface. Several important non-class library functions are provided. A basic linked list class called SequentialList is heavily used. States (see section 9.1) and Portholes (see section 6.2) can have *attributes*; these are particularly important in code generation. Finally, many of the significant classes in Ptolemy – functional blocks, portholes to implement connections, parameters – are derived from NamedObj, the basic object for implementing a named object that lives in a hierarchy.

1.1 The C++ Subset Used In Ptolemy

The Ptolemy system has grown up with the C++ language, so it does not use all the latest features in the newest compilers or every nook and cranny of Ellis and Stroustrup's Annotated Reference Manual, because of unimplemented features or lack of stability of implementation. Instead, we focused on stability. Accordingly, Ptolemy can be built with a number of different C++ compilers. This means, for one thing, that templates are not used (except in the experimental IPUS domain). In addition, some features that do not work that well yet under g++, such as nested classes, are also avoided. Nested enumerations, however, are used in several places.

1.2 Iterators

Iterators are a very basic and widely used concept in Ptolemy, and are used repeatedly in Ptolemy programming in almost any situation where a composite object contains other objects. We have chosen to use a consistent interface for all iterator objects. The typical iterator class has the following basic interface (some iterators provide additional functions as well):

```
class MyIterator {
public:
// constructor: argument is associated outer object
      MyIterator(OuterObject&);
      // next: return a pointer to the next object,
      // or a null pointer if no more
      InnerObject* next();
      // operator form: a synonym for next
      InnerObject* operator++(POSTFIX_OP) {return next();}
      // reset the iterator to point to the first object
      void reset();
}
```

`POSTFIX_OP` is a macro that is defined to be an empty string on older compilers (such as cfront 2.1 and versions of g++ before 2.4) and to the string "int" with newer compilers. This conditional behavior is required because of the evolution of the C++ language; previously,

postfix and prefix forms of the operators ++ and – were not distinguished when overloaded; now, a dummy int argument indicates that the postfix form is intended.

A typical programming application for iterators might be something like

```
// print the names of all objects in the container
ListIter nextItem(myList);
Item *itemP;
while ((itemP = nextItem++) != 0)
      cout << itemP->name() << "\back n";
```

It is, as a rule, not safe to modify most container classes in parallel with the use of an iterator, as the iterator may attempt to access an object that does not exist any more. However, the `reset` member function will always make the iterator safe to use even if the list has been modified (user-written iterators should preserve this property).

1.3 Non-class utility procedures

The kernel provides several useful ordinary (non-class) procedures, primarily for manipulating strings and path names. Some are defined in `miscFuncs.h,` others in `paths.h.`

`char* savestring(const char* text);`
Create a copy of the *`text`* argument with `new` and return a pointer to it. It is the caller's responsibility to assure that the string is eventually deleted by using the `delete` [] operator. The argument *`text`* must not be a null pointer.

`const char* hashstring(const char* text);`
Enters a copy of *`text`* into a hash table and return a pointer to the entry. If two strings compare equal when passed to `strcmp,` then if both are passed to `hashstring,` the return values will be the same pointer.

`const char* expandPathName(const char* fileName);`
Expand a path name that may begin with an environment variable or a user's home directory. If the string does not begin with a ~ or $ character, the string itself is returned. A leading "`~/`" is replaced by the user's home directory; a leading "~*`user`*" is replaced by the home directory for *`user,`* unless there is no such user, in which case the original string is returned. Finally, a leading "$*`env`*" is replaced by the value of the environment variable *`env;`* if there is no such environment variable, the original string is returned. Note that references to environment variables other than at the beginning are *not* substituted. If any substitutions are made, the return value is actually a pointer into a static buffer. This means that a second call to this function may write on top of a value returned by a previous call.

`const char* pathSearch(const char* file, const char* path=0);`
For this function, *`path`* is a series of Unix-style directory names, separated by colons. If no second argument is supplied or if the value is null, the value of the PATH environment variable is used instead. For each of the colon-separated directory strings, the function checks to see whether the file exists in the named directory. If it finds a match, it returns a pointer to an internal buffer containing the full path of the match. If it does not find a match, it returns a null pointer.

`int progNotFound(const char* program,const char* extra=0);`
This function searches for *program* in the user's PATH using the `pathSearch` function. If a match is found, the function returns false (0). Otherwise it returns true (1) and also generates an error message with the `Error::abortRun` function. If the *extra* argument is given, it forms the second line of the error message.

1.4 Generic Data Structures

As Ptolemy does not use templates, our generic lists use the generic pointer technique, with

```
typedef void * Pointer;
```

The most commonly used generic data structure in Ptolemy is `SequentialList.` Other lists are, as a rule, privately inherited from this class, so that type safety can be preserved. It is possible to insert and retrieve items at either the head or the tail of the list.

1.5 Class SequentialList

This class implements a single linked list with a count of the number of elements. The constructor produces a properly initialized empty list, and the destructor deletes the links. However, the destructor does not delete the items that have been added to the list; this is not possible because it has only `void *` pointers and would not know how to delete the items. There is an associated iterator class for SequentialList called ListIter.

1.5.1 SequentialList information functions

These functions return information about the SequentialList but do not modify it.

`int size() const;`
Return the size of the list.

`Pointer head() const;`
Return the first item from the list (0 if the list is empty). The list is not changed.

`Pointer tail() const;`
Return the last item from the list (0 if the list is empty). The list is not changed.

`Pointer elem(int n) const;`
Return the *nth* item on the list (0 if there are fewer than *n* items). Note that the time required is proportional to *n.*

`int empty() const;`
Return 1 if the list is empty, 0 if it is not.

`int member(Pointer arg) const;`
Return 1 if the list has a Pointer that is equal to *arg,* 0 if not.

1.5.2 Functions that modify a SequentialList

`void prepend(Pointer p);`
Add an item at the beginning of the list.

`void append(Pointer p);`
Add an item at the end of the list.

`int remove(Pointer p);`
Remove the pointer *p* from the list if it is present (the test is pointer equality). Return 1 if present, 0 if not.

`Pointer getAndRemove();`
Return and remove the head of the list. If the list is empty, return a null pointer (0).

`Pointer getTailAndRemove();`
Return and remove the last item on the list.

`void initialize();`
Remove all links from the list. This does not delete the items pointed to by the pointers that were on the list.

1.5.3 Class ListIter

ListIter is a standard iterator class for use with objects of class SequentialList. The constructor takes an argument of type `const SequentialList` and the ++ operator (or `next` function) returns a `Pointer.` Class ListIter is a friend of class SequentialList. In addition to the standard iterator functions `next` and `reset,` this class also provides a function

```
void reconnect(const SequentialList& newList)
```

that attaches the ListIter to a different SequentialList.

1.6 Doubly linked lists

Support for doubly linked lists is found in `DoubleLink.h.` The class DoubleLink implements a base class for nodes in the list, class DoubleLinkList implements the list itself, and class DoubleLinkIter forms an iterator. *WARNING: We consider this class to have serious design flaws, so it may be reworked quite a bit in subsequent Ptolemy releases.*

1.6.1 Class DoubleLink

A DoubleLink object is an item in the list defined by DoubleLinkList. Normally, a programmer will not interact directly with this class, but rather will use methods in DoubleLinkList. Nonetheless, we present it here because some of the methods of DoubleLinkList do refer to it.

There are two constructors:

`DoubleLink(Pointer p, DoubleLink* next, DoubleLink* prev`):
`DoubleLink(Pointer p);`
The first form initializes the *next* and *prev* pointers of the node as well as the contents. The second form sets these pointers to null and only initializes the contents pointer.

`Pointer content();`
Return the content pointer of the node.

`virtual ~DoubleLink();`
The destructor is virtual.

`void unlinkMe();`
Delete the node from the list it is contained in. I.e. connect the elements pointed to by the *prev* and *next* pointers. The object pointed to by the node is not deleted.

The following data members are protected:

```
DoubleLink *next; // next node in the list
DoubleLink *prev; // previous node in the list
Pointer e;        // contents of this node
```

1.6.2 Class DoubleLinkList

`DoubleLinkList();`
`DoubleLinkList(Pointer* e);`
The first constructor creates an empty list. The second creates a one-node list containing the object pointed to by *e*. That object must live at least as long as the link lives.

`virtual ~DoubleLinkList();`
The destructor is virtual. It deletes all DoubleLinks in the list, but does not delete the objects pointed to by each link.

`DoubleLink* createLink(Pointer e);`
Return a newly allocated DoubleLink that contains a pointer to *e*. It is up to the caller to delete the DoubleLink, or to use either `removeLink` or `remove`.

`void insertLink(DoubleLink *x);`
`void insert(Pointer e);`
These methods insert an item at the beginning of the list. The first inserts a DoubleLink; the second creates a DoubleLink with `createLink` and inserts that. If the second form is used, the link should only be removed using `removeLink` or `remove`, not `unlink`, because `unlink` will not delete the DoubleLink.

`void appendLink(DoubleLink *x);`
`void append(Pointer e);`
These methods append at the end of the list. The first appends a DoubleLink; the second creates a DoubleLink with `createLink` and appends that. If the second form is used, the link should only be removed using `removeLink` or `remove`, not `unlink`, because `unlink` will not delete the DoubleLink.

`void insertAhead(DoubleLink *y, DoubleLink *x);`
`void insertBehind(DoubleLink *y, DoubleLink *x);`
The first method inserts *y* immediately ahead of the DoubleLink pointed to by *x*; the second inserts *y* immediately after the DoubleLink pointed to by *x*. Both of these functions assume that `x` is in the list; disaster may result otherwise.

`DoubleLink* unlink(DoubleLink *x);`
Remove the link *x* from the list and return a pointer to it. Make sure that *x* is in the list

before calling this method, or disaster may result.

`void removeLink(DoubleLink *x);`
This is the same as `unlink,` except that *x* is deleted as well. The same cautions apply.

`void remove(Pointer e);`
Search for a DoubleLink whose contents match *e*. If a match is found, the node is removed from the list and the DoubleLink is deleted. The object pointed to by *e* is not deleted. The search starts at the head of the list.

`int find(Pointer e);`
Search for a DoubleLink whose contents match *e*. If a match is found, 1 (true) is returned; otherwise 0 (false) is returned. The search starts at the head of the list.

`virtual void initialize();`
Delete all DoubleLinks in the list and make the list empty.

`void reset();`
Make the list empty, but do not delete the DoubleLinks in each of the nodes.

`int size();`
Return the number of elements in the list. This method should be const but isn't.

```
DoubleLink *head();
DoubleLink *tail();
```
Return a pointer to the head or to the tail of the list. If the list is empty both methods will return a null pointer.

```
DoubleLink *getHeadLink();
Pointer takeFromFront();
```
The first method gets and removes the head link, returning a pointer to it. The second method returns the object pointed to by the head link, and deletes the DoubleLink. If the list is empty, both functions return a null pointer.

```
DoubleLink *getTailLink();
Pointer takeFromBack();
```
These methods are identical to the previous pair except that they remove the last node rather than the first.

The following two data members are protected:

```
DoubleLink *myHead;
DoubleLink *myTail;
```

1.6.3 Class DoubleLinkIter

DoubleLinkIter is an iterator for DoubleLinkList. It is only capable of moving "forward" through the list (following the "next" links, not the "prev" links). Its `next` operator returns the

Pointer values contained within the nodes; it is also possible to use the non-standard `nextLink` function to return successive DoubleLink pointers.

1.7 Other generic container classes

The file `DataStruct.h` defines two other generic container classes that are privately derived from SequentialList: Queue and Stack. Class Queue may be used to implement a FIFO or a LIFO queue, or a mixture. Class Stack implements a stack.

1.7.1 Class Queue

The constructor for class Queue builds an empty queue. The following four functions move pointers into or out of the queue:

```
void putTail(Pointer p);
void putHead(Pointer p);
Pointer getHead();
Pointer getTail();
```

In addition, `put` is a synonym for `putTail`, and `get` is a synonym for `getHead`. All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions `size` and `initialize` are re-exported (that is, are accessible as public member functions of class Stack).

1.7.2 Class Stack

The constructor for class Stack builds an empty stack. The following functions move pointers onto or off of the stack:

```
void pushTop(Pointer p);
Pointer popTop();
pushBottom(Pointer p);
```

`pushTop` and `popTop` are the functions traditionally associated with a stack; `pushBottom` adds an item at the bottom, which is non-traditional. The following non-destructive function also exists:

```
Pointer accessTop() const;
```

This accesses but does not remove the element from the top of the stack.

All these functions are implemented on top of the (hidden) SequentialList functions. The SequentialList functions `size` and `initialize` are re-exported.

1.8 Class NamedObj

NamedObj is the base class for most of the common Ptolemy objects. A NamedObj is, simply put, a named object; in addition to a name, a NamedObj has a pointer to a parent object, which is always a Block (a type of NamedObj). This pointer can be null. A NamedObj also has a descriptor. Warning! NamedObj assumes that the name and descriptor “live” as long as the NamedObj does. They are not deleted by the destructor, so that they can be compile-time strings. Important derived types of NamedObj include Block (see section3.1), GenericPort (see section 6.1), State (see section 9.1), and Geodesic (see section 6.6).

1.8.1 NamedObj constructors and destructors

All constructors and destructors are public. NamedObj has a default constructor, which sets the name and descriptor to empty strings and the parent pointer to null, and a three-argument constructor:

`NamedObj(const char* name,Block* parent, const char* descriptor)`

NamedObj's destructor is virtual and does nothing.

1.8.2 NamedObj public members

`virtual const char* className() const;`
Return the name of the class. This needs to have a new implementation supplied for every derived class (except for abstract classes, where this is not necessary).

`const char* name() const;`
Return the local portion of the name of the class (vs. the full name).

`const char* descriptor() const;`
Return the descriptor.

`Block* parent() const;`
Return a pointer to the parent block.

`virtual StringList fullName() const;`
Return the full name of the object. This has no relation to the class name; it specifies the specific instance's place in the universe-galaxy-star hierarchy. The default implementation returns names that might look like `universe.galaxy.star.port` for a porthole; this is the full name of the parent, with a period and the name of the object appended.

`void setName(const char* name);`
Set the name of the object. The string must live at least as long as the object.

`void setParent(Block* parent);`
Set the parent of the object, which is always a Block. The parent must live at least as long as the object.

`void setNameParent (const char* my_name, Block* my_parent)`
Change the name and parent pointer of the object.

`virtual void initialize() = 0;`
Initialize the object to prepare for system execution. This is a pure virtual method.

`virtual StringList print (int verbose) const;`
Return a description of the object. If the argument *verbose* is 0, a somewhat more compact form is printed than if the argument is non-zero.

`virtual int isA(const char* cname) const;`
Return TRUE if the argument is the name of the class or of one of its base classes. This

method needs to be redefined for all classed derived from NamedObj. To make this easy to do, a macro `ISA_FUNC` is provided; for example, in the file `Block.cc` we see the line

```
ISA_FUNC(Block,NamedObj);
```

NamedObj is the base class from which Block is derived. This macro creates the function definition

```
int Block::isA(const char* cname) const {
if (strcmp(cname,"Block") == 0) return TRUE;
      else return NamedObj::isA(cname);
}
```

Methods `isA` and `className` are overridden in all derived classes; the redefinitions will not be described for each individual class.

1.8.3 Flags on named objects

`FlagArray flags`

Many schedulers and targets need to be able to mark blocks in various ways, to count invocations, or flag that the block has been visited, or to classify it as a particular type of block. To support this, we provide an array of flags that are not used by class Block, and may be used in any way by a Target. The target may defer their use to its schedulers. The array can be of any size, and the size will be increased automatically as elements are referenced. For readability and consistency, the user should define an enum in the target or scheduler class to give the indices, so that mnemonic names can be associated with flags, and so that multiple schedulers for the same target are consistent. For instance, if `b` is a pointer to a `Block,` a target might contain the following:

```
private:
      enum {
            visited = 0,
            fired = 1
            }
```

which can then be used in code to set and read flags in a readable way,

```
b->flags[visited] = TRUE;
   ...
if (b->flags[visited]) { ... }
```

WARNING: For efficiency, there is no checking to prevent two different pieces of code (say a target and scheduler) from using the same flags (which are indexed only by non-negative integers) for different purposes. The policy, therefore, is that *the target is in charge*. It is incumbent upon the writer of the target to know what flags are used by schedulers invoked by that target, and to avoid corrupting those flags if the scheduler needs them preserved. We weighed a more modular, more robust solution, but ruled in out in favor of something very lightweight and fast.

1.8.4 NamedObj protected members

`void setDescriptor(const char* desc);`

Set the descriptor to *desc*. The string pointed to by *desc* must live as long as the NamedObj does.

1.9 Class NamedObjList

Class NamedObjList is simply a list of objects of class NamedObj. It is privately inherited from class SequentialList (see section 1.5), and, as a rule, other classes privately inherit from it. It supports only a subset of the operations provided by SequentialList; in particular, objects are added only to the end of the list. It provides extra operations, like searching for an object by name and deleting objects. This object enforces the rule that only const pointers to members can be obtained if the list is itself const; hence, two versions of some functions are provided.

1.9.1 NamedObjList information functions

The `size` and `initialize` functions of SequentialList are re-exported. Note that `initialize` removes only the links to the objects and does not delete the objects. Here is what's new:

```
NamedObj* objWithName(const char* name);
const NamedObj* objWithName(const char* name) const;
```

Find the first NamedObj on the list whose name is equal to *name,* and return a pointer to it. Return 0 if it is not found. There are two forms, one of which returns a const object.

```
NamedObj* head();
const NamedObj* head() const;
```

Return a pointer to the first object on the list (0 if none). There are two forms, one of which returns a const object.

1.9.2 Other NamedObjList functions

`void put(NamedObj& obj)`

Add a pointer to *obj* to the list, at the end. The object must live at least as long as the list.

`void initElements();`

Apply the `initialize` method to each NamedObj on the list.

`int remove(NamedObj* obj);`

Remove *obj* from the list, if present (this does not delete *obj).* Return 1 if it was present, 0 if not.

`void deleteAll();`

Delete all elements from the list, and reset it to be an empty list. WARNING: this assumes that the members of the list are on the heap (allocated by `new,` so that deleting them is valid)!

1.9.3 NamedObjList iterators

There are two different iterators associated with NamedObjList; class NamedObjListIter and class CNamedObjListIter. The latter returns const objects (which cannot then be modified). The former returns a non-const pointer, and can only be used if the NamedObjList itself is not const. Both obey the standard iterator interface and are privately derived from class ListIter.

1.10 Attributes

Attributes represent logical properties that an object may or may not have. Certain classes such as State and Porthole contain attributes and provide interfaces for setting and clearing attributes. For the State class, for instance, the initial value may or may not be settable by the user; this is indicated by an Attribute. In code generation classes, attributes may indicate whether an assembly-language buffer should be allocated to ROM or RAM, fast memory or slow memory, etc. The set of attributes of an object is stored in an entity called a `bitWord`. At present, a bitWord is represented as an unsigned long, which restricts the number of distinct attributes to 32; this may be changed in future releases. An Attribute object represents a request to turn certain attributes of an object off, and to turn other attributes on. As a rule, constants of class Attribute are used to represent attributes, and users have no need to know whether a given property is represented by a true or false bit in the bitWord. Although we would prefer to have a constructor for Attribute objects of the form

```
Attribute(bitWord bitsOn, bitWord bitsOff);
```

it has turned out that doing so presents severe problems with order of construction, since a number of global Attribute objects are used and there is no simple, portable way of guaranteeing that these objects are constructed before any use. As a result, the `bitsOn` and `bitsOff` members are public, but we forbid use of that fact except in one place: constant Attribute objects can be initialized by the C “aggregate form”, as in the following example:

```
extern const Attribute P_HIDDEN = {PB_HIDDEN, 0};
```

(This particular attribute is used by Porthole to indicate that a port should not be visible to the user, i.e. should not appear on an icon.) The first word specified is the `bitsOn` field, `PB_HIDDEN`, and the second word specified is the `bitsOff` field. Other than to initialize objects, we pretend that these data members are private.

1.10.1 Attribute member functions

```
Attribute& operator |= (const Attribute& arg);
Attribute& operator &= (const Attribute& arg);
```

These operations combine attributes, by applying the `|=` and `&=` operators to the bitsOn and bitsOff fields. The first operation, as attributes are commonly used, represents a requirement that two sets of attributes be met, so it has been argued that it really should be the “and” operation. However, the current scheme has the virtue of consistency.

```
bitWord eval(bitWord defaultVal) const;
```

Evaluate an attribute given a default value. Essentially, bits corresponding to bitsOn are turned on, and then bits corresponding to bitsOff are turned off.

`bitWord clearAttribs(bitWord` *`defaultVal`*`) const;`
This method essentially applies the attribute backwards, reversing the roles of bitsOn and bitsOff in `eval`.

```
bitWord on() const;
bitWord off() const;
```
Retrieve the bitsOn and bitsOff values, respectively. Inline definitions of operators `&` and `|` are also defined to implement nondestructive forms of the `&=` and `|=` operations.

1.11 FlagArray

`FlagArray` is a lightweight, self-expanding array of integers. It is meant to store an array of flags or counters, and its main appearance in Ptolemy is as a public member of class `NamedObj`, and therefore is available in most Ptolemy classes, which are derived from `NamedObj`. Targets and schedulers use this member to keep track of various kinds of data. Many schedulers and targets need to be able to mark blocks in various ways, for example to count invocations, or flag that the block has been visited, or to classify it as a particular type of block. This class provides a simple mechanism for doing this. A `FlagArray` object is indexed like an array, using square brackets. If `x` is a `FlagArray` and `i` is a non-negative integer, then `x[i]` is a reference to an integer element of the array. If `i` is out of bounds (beyond the currently allocated limits of the array), then the class automatically increases the size of the array. New elements are filled with zeros. Thus, a `FlagArray` may be viewed as an infinite dimensional array of integers initialized with zeros. If `i` is a negative integer, then `x[i]` is an error. For efficiency, the class does not test for this error at run time, so you could get a core dump if you make this error.

1.11.1 FlagArray constructors and destructor

`FlagArray()`
This constructor creates a zero-length flag array.

`FlagArray(int size)`
This constructor creates a flag array with the specified size already allocated and filled with zeros.

`FlagArray(int` *`size`*`, int` *`fill_value`*`)`
This constructor creates a flag array with the specified size filled with the specified integer value. The destructor frees the memory allocated to store the array of integers.

1.11.2 FlagArray public methods

`FlagArray & operator = (const FlagArray &` *`v`*`)`
An assignment to one `FlagArray` from another simply copies its size and data.

`int size() const`
Return the current allocated size of the array.

`int & operator [] (int` *`n`*`)`
If *n* is less than the currently allocated size of the array, then this returns a reference to the

n-th element of the array. If *n* is greater than or equal to the currently allocated size of the array, then the size of the array is increased, the new elements are filled with zeros, and a reference to the n-th element is returned. Indexing of elements begins with zero. The returned reference, of course, can be used on the left-hand side of an assignment. This is how values are written into an array.