Chapter 2. Writing Stars for Simulation

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

Other Contributors: *Most of the Ptolemy team*

2.1 Introduction

Ptolemy provides rich libraries of stars for the more mature domains. Since the stars were designed to be as generic as possible, many complicated functions can be realized by a galaxy. Nonetheless, no star library can possibly be complete; you may need to design your own stars. The Ptolemy preprocessor language makes this easier than it could be. This chapter is devoted to the use of the preprocessor language.

Newly designed stars can be dynamically linked into Ptolemy, avoiding frequent recompilation of the system. If the new stars are generic and useful, however, it might be better to add them to the list of compiled-in stars and rebuild the system. See "Creating Custom Versions of pigiRpc" on page 1-6.

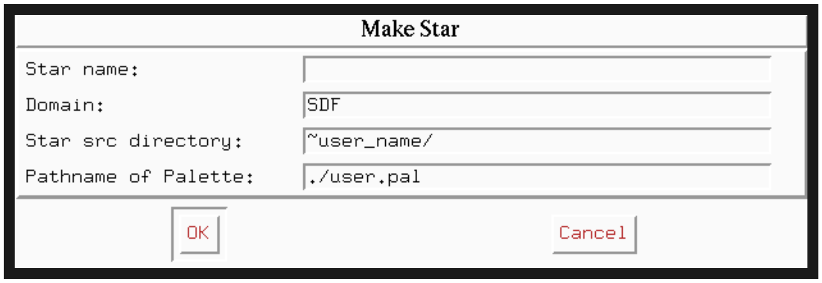
2.2 Adding stars dynamically to Ptolemy

To get a quick sense of what it means to create a new star, you can use one of the existing stars as a template. Create a new directory in which you have write permission. Copy the source code for an existing Ptolemy star. For example,

```
cd my_directory
cp $PTOLEMY/src/domains/sdf/stars/SDFSin.pl SDFMyStar.pl
chmod +w SDFMyStar.pl
```

The "`.pl`" extensions on the file names stand for "Ptolemy language" or "preprocessor language." The file name must be of the form `*DomainStarname*.pl` for dynamic linking and the `look-inside` command to work. The last command just ensures that you can modify the file. Edit the file to change the name of the star from `Sin` to `MyStar`. This is necessary so that the name does not conflict with the existing `Sin` star in the SDF domain.

You can now dynamically link your new star. Start `pigi`, the graphical editor. If you start `pigi` in your new directory, you will get a blank `init.pal` facet. Place your mouse cursor in this facet, and issue the "make-star" command (the shortcut is "`*`"). A dialog box like the following will appear:



Enter the name of the star, `MyStar`, its domain, `SDF`, the location of the directory that defines it, such as `~user_name/my_directory`, and the name of palette in which you would like its icon to appear, `user.pal`. The star will be compiled and dynamically linked with the Ptolemy executable. An icon for it will appear in the facet `user.pal`. Try using this in a simple system.

Three details about dynamic linking may prove useful:

- If the name of the star source directory has a `/src/` component, `pigi` will replace this with `/obj.$PTARCH/` depending on the type of machine you are running, to get the name of the directory in which to store the object file. This is especially useful if you are jointly doing development with others who use a different type of machine. If there is no `/src/` component in the name, then the object file is placed in the same directory with the source file.
- If there is a file named `Makefile` or `makefile` in the object file directory, pigi will run the `make` program, using the `makefile` to create the object file (or make sure it is up to date). If there is no `makefile`, pigi will run a make-like procedure on its own, running the preprocessor as needed to produce the C++ source files, then running the C++ compiler to create the object file. By default, the C++ compiler will be told to look for include files in the kernel directory and the domain-specific kernel and star directories; if this is not adequate, then you need to write a makefile. Once compilation (if any) is complete, the dynamic linker is used to load the star into the system. Compilation errors, if any, will appear in a popup window.
- Whenever the definition of a star is changed so that the new definition has different I/O ports, the icon must be updated as well. You can do this by calling *make-star* again to replace the old icon with a new one.

If the linking fails, one of the following situations may apply:

- Whoever installed Ptolemy did not install the compiler.
- The compiler is not configured correctly. If you are using a prebuilt compiler obtained from the Ptolemy ftp site, you may need to set some environment variables if your Ptolemy installation is not at `/users/ptolemy`. See Appendix A of the Ptolemy *User's Manual* for more information.
- A spurious `makefile` exists in your directory. If a makefile exists in your directory, Ptolemy will attempt to use it to compile your star. Remove it, and try again.
- The version of the compiler used to build Ptolemy is not the same as the version

used to compile your star. This should not occur if you are using the compiler distributed with Ptolemy, but can occur if the compiler has been updated since Ptolemy was last built, or if you are not using the compiler distributed with Ptolemy.

- You have a `/src/` component in the directory name, but the corresponding `/obj.$PTARCH/` directory does not exist or cannot be written.

You may find it helpful to refer to the Appendix A, Installation and Troubleshooting in the *User's Manual*.

The star you just created performs exactly the same function as an existing star in the Ptolemy library, and hence is not very interesting. Try modifying the star. For example, you could add 1.0 to the sine before producing the output. Find the definition of the `go` method, which should look like this:

```
go {
      output%0 << sin (double(input%0));
}
```

The one line of code is ordinary C++ code, although the "<<" and "`%`" operators have been overloaded. This line means that the current value (`%0`) of the output named "`output`" should be assigned the value returned by the `sin` function applied to the current value of the input named "`input`". The cast to `double` indicates that we are not really interested in the `Particle` object supplied by the input, but rather its value, interpreted as a double-precision floating point number. Try changing this code to

```
go {
      output%0 << sin (double(input%0)) + 1.0;
}
```

To recompile and reload the star, place your mouse cursor on any instance of the icon for the star, and type "`L`" (or invoke the "Extend:load-star" command through the menus).

Sometimes, you will wish to dynamically link stars that are derived from other stars that you have dynamically linked. To do this, the base class stars must be *permanently linked*. This can be done with the "Extend:load-star-perm" command ("`K`"). To do this, place the mouse over an icon representing the parent star, and type "`K`". Once the parent star is permanently linked, it cannot be replaced or redefined: you must restart `pigi`.

The `go` and all other entries in the `.pl` file defining the star are explained in the following sections.

2.3 The Ptolemy preprocessor language (ptlang)

The Ptolemy preprocessor was created to make it easier to write and document star class definitions to run under Ptolemy. Instead of writing all the class definitions and initialization code required for a Ptolemy star, the user can concentrate on writing the action code for a star and let the preprocessor generate the standard initialization code for portholes, states, etc. The preprocessor generates standard C++ code, divided into two files (a header file with a "`.h`" extension and an implementation file with a "`.cc`" extension). It also generates standardized documentation, in a file with a "`.t`" extension, to be included in the manual. The `.t` file is troff source for documentation.

2.3.1 Invoking the preprocessor

The definition of a star named YYY in domain XXX should appear in file with the name XXXYYY.pl. The class that implements this star will be named XXXYYY. Then, running the command

```
ptlang XXXYYY.pl
```

will produce the files XXXYYY.cc, XXXYYY.h, and XXXYYY.t. The documentation file XXX-YYY.t can be printed using the command $PTOLEMY/doc/bin/printpage. The print-page command assumes your PRINTER environment variable is set to the printer of choice. The following line in your .cshrc file, for example, will ensure this:

```
setenv PRINTER printer_name
```

It also assumes that you have the ditroff program in your path.

2.3.2 Implementation of the preprocessor

The preprocessor is written in yacc and C. It does not attempt to parse the parts of the language that consist of C++ code (for example, go methods); for these, it simply counts curly braces to find the ends of the items in question. It outputs #line directives so the C++ compiler will print error messages, if any, with respect to the original source file. It also makes no attempt to parse information in the explanation directive, so arbitrary troff formatting commands can be used (see the "Documentation" chapter in the *User's Manual).* It can also contain eqn directives (for formatting equations) and tbl directives (for creating tables). These are standard Unix tools.

2.3.3 An example

To make things clearer, let us start with an example, a rectangular pulse star in the file SDFRect.pl:

```
defstar {
      name { Rect }
      domain { SDF }
      desc {
Generates a rectangular pulse of height "height" (default 1.0).
with width "width" (default 8).
      }
      version {%W% %G%}
      author { J. T. Buck }
      copyright {1993 The Regents of the University of California}
      location { SDF main library }
      state {
            name { height }
            type { float }
            default { 1.0 }
            desc { Height of the rectangular pulse. }
      }
      state {
            name { width }
            type { int }
            default { 8 }
            desc { Width of the rectangular pulse. }
```

```
        }
        state {
              name { count }
              type { int }
              default { 0 }
              desc { Internal counting state. }
              attributes { A_NONSETTABLE|A_NONCONSTANT }
        }
        output {    // the output port
              name { output }
              type { float }
              desc { The output pulse. }
        }
        go {              // the run-time function
              double t = 0.0;
              if (count < width) t = height;
              count = int(count) + 1;
              output%0 << t;
        }
  }
```

Running the preprocessor on the above file produces the three files `SDFRect.h`, `SDFRect.cc` and `SDFRect.t`; the names are determined *not* by the input filename but by concatenating the domain and name fields. These files define a class named `SDFRect`.

At the time of this writing, only one type of declaration may appear at the top level of a Ptolemy language file, a `defstar`, used to define a star. Sometime in the future, a `defgalaxy` section may also be supported. The `defstar` section is itself composed of subitems that define various attributes of the star. All subitems are of the form

keyword { *body* }

where the *body* may itself be composed of sub-subitems, or may be C++ code (in which case the Ptolemy language preprocessor checks it only for balanced curly braces). Note that the keywords are *not* reserved words; they may also be used as identifiers in the body.

2.3.4 Items that appear in a defstar

The following items can appear in a `defstar` directive. The items are given in the order in which they typically appear in a star definition (although they can appear in any order). An alphabetical listing and summary of directives is given in table 2-1.

name

This is a required item, and has the syntax

name { *identifier* }

It (together with the domain) provides the name of the class to be defined and the names of the output files. Case is important in the identifier.

domain

This is a required item; it specifies the domain, such as SDF. The syntax is:

keyword	summary	required	page
`acknowl-edge`	the names of other contributors to the star	no	2-8
`author`	the name(s) of the author(s)	no	2-8
`begin`	C++ code to execute at start time, *after* the scheduler setup	no	2-14
`ccinclude`	specify other files to include in the .cc file	no	2-15
`code`	C++ code to include in the .cc file outside the class definition	no	2-15
`codeblock`	define a code segment for a code-generation star	no	12-2
`conscalls`	define constructor calls for members of the star class	no	2-13
`construc-tor`	C++ code to include in the constructor for the star	no	2-13
`copyright`	copyright information to include in the generated code	no	2-8
`derived`	alternative form of `derivedFrom`	no	2-7
`derived-from`	the base class, which must also be a star	no	2-7
`desc`	alternative form of `descriptor`	no	2-7
`descriptor`	a short summary of the functionality of the star	no	2-7
`destructor`	C++ code to include in the destructor for the star	no	2-13
`domain`	the domain, and the prefix of the name of the class	yes	2-5
`explana-tion`	full documentation, optionally using troff, eqn, and tbl formats	no	2-9
`exectime`	specify the execution time for a code generation star	no	12-2
`go`	C++ code to execute when the star fires	no	2-14
`header`	C++ code to include in the .h file, before the class definition	no	2-16
`hinclude`	specify other files to include in the .h file	no	2-15
`inmulti`	define a set of inputs	no	2-11
`inout`	define a (bidirectional) input and output	no	2-11
`inoutmulti`	define a set of (bidirectional) inputs and outputs	no	2-11
`input`	define an input to the star	no	2-11
`location`	an indication of where a user might find the star	no	2-9
`method`	define a member function for the star class	no	2-16
`name`	the name of the star, and the root of the name of the class	yes	2-5
`outmulti`	define a set of outputs	no	2-11
`output`	define an output from the star	no	2-11
`private`	define private data members of the star class	no	2-15
`protected`	defined protected data members of the star class	no	2-15
`public`	define public data members of the star class	no	2-15
`setup`	C++ code to execute at start time, *before* the scheduler setup	no	2-14
`state`	define a state or parameter	no	2-9
`version`	version number and date	no	2-8
`wrapup`	C++ code to invoke at the end of a run (if no error occurred)	no	2-14

TABLE 2-1: A summary of the items used to define a star. Additional items are allowed in code generation stars, as explained in later chapters. A minimal set of the most useful items are shaded.

```
domain { identifier }
```

where *identifier* specifies the domain (again, case is important). All standard Ptolemy domains are supported, although for the Thor domain, a separate preprocessor is normally used.

derivedfrom

This optional item indicates that the star is derived from another class. Syntax:

```
derivedfrom { identifier }
```

where *identifier* specifies the base class. The `.h` file for the base class is automatically included in the output `.h` file, assuming it can be located (you may need to create a makefile).

For example, the `LMS` star in the `SDF` domain is derived from the `FIR` star. The full name of the base class is `SDFFIR`, but the `derivedfrom` statement allows you to say either

```
derivedfrom { FIR }
```

or

```
derivedfrom { SDFFIR }
```

The `derivedfrom` statement may also be written `derivedFrom` or `derived`.

descriptor

This item defines a short description of the class. This description is displayed by the `profile` pigi command. It has the syntax

```
descriptor { text }
```

where *text* is simply a section of text that will become the short descriptor of the star. You may also write `desc` instead of `descriptor`. A principal use of the short descriptor is to get on-screen help, so the descriptor should not include any troff formatting commands. Unlike the `explanation` (described below), it does not pass through troff. The following are legal descriptors:

```
desc { A one line descriptor. }
```

or

```
desc {
A multi-line descriptor. The same line breaks and spacing
will be used when the descriptor is displayed on the screen.
}
```

By convention, in these descriptors, references to the names of states, inputs, and outputs should be enclosed in quotation marks. Also, each descriptor should begin with a capital letter, and end with a period. If the descriptor seems to get long, augment it with the `explanation` directive, explained below. However, it should be long enough so that it is sufficient to explain the function of the star.

version

This item contains two entries as shown below

```
version { number MO/DA/YR }
```

where the *number* is a version number, and the *MO/DA/YR* is the version date. If you are using SCCS for version control then the following syntax will work well:

```
version { %W% %G% }
```

When the file is checked in by SCCS, the string `%W%` will be replaced with a string of the form: `@(#)`*filename num*, where num is the version number, and `%G%` will be replaced with a properly formatted date.

author

This optional entry identifies the author or authors of the star. The syntax is

```
author { author1, author2 and author3 }
```

Any set of characters between the braces will be interpreted as a list of author names.

acknowledge

This optional entry attaches an acknowledgment section to the documentation. The syntax is

```
acknowledge { arbitrary single line of text }
```

copyright

This optional entry attaches a copyright notice to the `.h`, `.cc`, and `.t` files. The syntax is

```
copyright { copyright information }
```

For example, we used to use the following (our lawyers have recently caused us to increase the verbosity):

```
copyright {1994 The Regents of the University of California}
```

The copyright may span multiple lines, just like a descriptor. In house, we use the SCCS `%Q%` keyword to update the date when a file is changed. A typical copyright line might look like:

```
copyright {1990-%Q% The Regents of the University of
      California}
```

location

This item describes the location of a star definition. The following descriptions are used, for example:

```
location { SDF dsp library }
```

or

```
location { directory }
```

where *directory* is the location of the star. This item is for documentation only.

explanation

This item is used to give longer explanations of the function of the stars. The explanation will appear only in the `.t` troff documentation file. The syntax is

```
explanation {
body
}
```

The body is assumed to be troff source, but a number of macros are provided for convenience. The most frequently used macro in `explanation` sections is the un-numbered header, invoked as follows:

```
.UH "Header title"
```

Star inputs, outputs, and state names, by convention, should be given in *italics*. This can be conveniently done using the font-change escapes, as in “\fIinput\fP”. Ptolemy class names are, by convention, set in a special font. This font may be obtained in-line by the escape sequence “\f(*F”, and again the previous font can be restored using “\fP”. Alternatively, the font can be obtained using the troff macro:

```
.c word
```

or the start and end macros

```
.(c
body
.)c
```

More formatting options are given in the “Documentation” chapter in the *User’s Manual*.

state

This item is used to define a state or parameter. Recall that by definition, a parameter is the initial value of a state. Here is an example of a state definition:

```
state {
      name { gain }
      type { int }
      default { 10 }
      desc { Output gain. }
      attributes { A_CONSTANT|A_SETTABLE }
}
```

There are five types of subitems that may appear in a state statement, in any order. The `name` field is the name of the state; the `type` field is its type, which may be one of `int`, `float`, `string`, `complex`, `fix`, `intarray`, `floatarray`, `complexarray`, `precision`, or `stringarray`. Case is ignored for the type argument.

The `default` item specifies the default initial value of the state; its argument is either a string (enclosed in quotation marks) or a numeric value. The above entry could equivalently have been written:

```
default { "1.0" }
```

Furthermore, if a particularly long default is required, as for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a `ComplexArray`:

```
default {
"(-.040609,0.0) (-.001628,0.0) (.17853,0.0) (.37665,0.0)"
"(.37665,0.0) (.17853,0.0) (-.001628,0.0) (-.040609,0.0)"
}
```

For complex states, the syntax for the default value is

```
(real, imag)
```

where `real` and `imag` evaluate to integers or floats.

The `precision` state is used to give the precision of fixed-point values. These values may be other states or may be internal to the star. The default can be specified in either of two ways:

- **Method 1**: As a string like "3.2", or more generally "*m.n*", where *m* is the number of integer bits (to the left of the binary point) and *n* is the number of fractional bits (to the right of the binary point). Thus length is *m*+*n*.
- **Method 2**: A string like "24/32" which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

The `desc` (or `descriptor`) item, which is optional but highly recommended, attaches a descriptor to the state. The same formatting options are available as with the star descriptor.

Finally, the `attributes` keyword specifies state attributes. At present, two attributes are defined for all states: `A_CONSTANT` and `A_SETTABLE` (along with their complements `A_NONCONSTANT` and `A_NONSETTABLE`). If a state has the `A_CONSTANT` attribute, then its value is not modified by the run-time code in the star (it is up to you as the star writer to ensure that this condition is satisfied). States with the `A_NONCONSTANT` attribute may change when the star is run. If a state has the `A_SETTABLE` attribute, then user interfaces (such as `pigi`) will prompt the user for values when directives such as *edit-parameters* are given. States without this attribute are not presented to the user; such states always start with their default values as the initial value. If no attributes are specified, the default is `A_CONSTANT|A_SETTABLE`. Thus, in the above example, the `attributes` directive is unnecessary. The notation "`A_CONSTANT|A_SETTABLE`" indicates a logical "or" of two flags. Confusingly, this means that they both apply (`A_CONSTANT` *and* `A_SETTABLE`).

Code generation stars use a great number of attributes, many specific to the language model for which code is being generated. Read chapter 12, "Code Generation", and the documentation for the appropriate code generation domain to learn more about these.

Mechanisms for accessing and updating states in C++ methods associated with a star are explained below, in sections 2.4.3 on page 2-22 and 2.4.4 on page 2-23.

An alternative form for the `state` directive is `defstate`. The subitems of the `state` directive are summarized in table 2-2, together with subitems of other directives.

input, output, inout, inmulti, outmulti, inoutmulti

These keywords are used to define a porthole, which may be an input, output, inout (bidirectional) porthole or an input, output, or inout multiporthole. Bidirectional ports are not supported in most domains (Thor is an exception). Like `state`, it contains subitems. Here is an example:

```
input {
      name { signalIn }
      type { complex }
      numtokens { 2 }
      desc {A complex input that consumes 2 input particles.}
}
```

Here, `name` specifies the porthole name. This is a required item. `type` specifies the particle type. The scalar types are `int`, `float`, `fix`, `complex`, `message`, or `anytype`. Again, case does not matter for the type value. The matrix types are `int_matrix_env`, `float_matrix_env`, `complex_matrix_env`, and `fix_matrix_env`. The `type` item may be omitted; the default type is `anytype`. For more information on all of these, please see chapter 4, "Data Types".

The `numtokens` keyword (it may also be written `num` or `numTokens`) specifies the number of tokens consumed or produced on each firing of the star. This only makes sense for certain domains (SDF, DDF, and BDF); in such domains, if the item is omitted, a value of one is used. For stars where this number depends on the value of a state, it is preferable to leave out the `numtokens` specification and to have the `setup` method set the number of tokens (in the SDF domain and most code generation domains, this is accomplished with the `setSDFParams` method). This item is used primarily in the SDF and code generation domains, and is discussed further in the documentation of those domains.

There is an alternative syntax for the type field of a porthole; this syntax is used in connection with `ANYTYPE` to specify a link between the types of two portholes. The syntax is

```
type { = name  }
```

where `name` is the name of another porthole. This indicates that this porthole inherits its type from the specified porthole. For example, here is a portion of the definition of the SDF `Fork` star:

item	sub-item	summary	required	page
inmulti, inout, inoutmulti, input	name	name of the port or group of ports	yes	11
	type	data type of input (& output) particles	no	
	descriptor	summary of the function of the input	no	
	numtokens	number of tokens consumed by the port (useful only for dataflow domains)	no	
method, virtual method, inline method, pure method, pure virtual method, inline virtual method	name	the name of the method	yes	16
	access	private, protected, or public	no	
	arglist	the arguments to the method	no	
	type	the return type of the method	no	
	code	C++ code defining the method	if not pure	
outmulti, output	name	name of the port or group of ports	yes	11
	type	data type of output particles	no	
	descriptor	summary of the function of the output	no	
	numtokens	number of tokens produced by the port (useful only for dataflow domains)	no	
state	name	the name of the state variable	yes	9
	type	data type of the state variable	yes	
	default	the default initial value, always a string	yes	
	descriptor	summary of the function of the state	no	
	attributes	hints to the simulator or code generator	no	

TABLE 2-2: Some items used in defining a star have subitems. These are described here.

```
input {
      name{input}
      type{ANYTYPE}
}
outmulti {
      name{output}
      type{= input}
      desc{ Type is inherited from the input. }
}
```

constructor

This item allows the user to specify extra C++ code to be executed in the constructor for the class. This code will be executed *after* any automatically generated code in the constructor that initializes portholes, states, etc. The syntax is:

```
constructor { body }
```

where `body` is a piece of C++ code. It can be of any length. Note that the constructor is invoked only when the class is first instantiated; actions that must be performed before every simulation run should appear in the `setup` or `begin` methods, not the constructor.

conscalls

You may want to have data members in your star that have constructors that require arguments. These members would be added by using the `public`, `private`, or `protected` keywords. If you have such members, the `conscalls` keyword provides a mechanism for passing arguments to the constructors of those members. Simply list the names of the members followed by the list of constructor arguments for each, separated by commas if there is more than one. The syntax is:

```
conscalls { member1(arglist), member2(arglist) }
```

Note that *`member1`,* and *`member2`* should have been previously defined in a `public`, `private`, or `protected` section (see page 2-15).

destructor

This item inserts code into the destructor for the class. The syntax is:

```
destructor { body }
```

You generally need a destructor only if you allocate memory in the constructor, `begin` method, or `setup` method; termination functions that happen with every run should appear in the `wrapup` function[1]. The optional keyword `inline` may appear before

1. Note, however, that wrapup is not called if an error occurs. See page 2-14.

destructor; if so, the destructor function definition appears inline, in the header file. Since the destructor for all stars is virtual, this is only a win when the star is used as a base for derivation.

setup

This item defines the setup method, which is called every time the simulation is started, *before* any compile-time scheduling is performed. The syntax is:

```
setup { body }
```

The optional keyword inline may appear before the setup keyword. It is common for this method to set parameters of input and output portholes, and to initialize states. The code syntax for doing this is explained starting on page 2-17. In some domains, with some targets, the setup method may be called more than once during initiation. You must keep this in mind if you use it to allocate or initialize memory.

begin

This item defines the begin method, which is called every time the simulation is started, but *after* the scheduler setup method is called (i.e., after any compile-time scheduling is performed). The syntax is:

```
begin { body }
```

This method can be used to allocate and initialize memory. It is especially useful when data structures are shared across multiple instances of a star. It is always called exactly once when a simulation is started.

go

This item defines the action taken by the star when it is fired. The syntax is:

```
go { body }
```

The optional keyword inline may appear before the go keyword. The go method will typically read input particles and write outputs, and will be invoked many times during the course of a simulation. The code syntax for the body is explained starting on page 2-17.

wrapup

This item defines the wrapup method, which is called at the completion of a simulation. The syntax is:

```
wrapup { body }
```

The optional keyword inline may appear before the wrapup keyword. The wrapup method might typically display or store final state values. The code syntax for doing this is explained starting on page 2-17. Note that the `wrapup` method is not invoked if an error occurs during execution. Thus, the `wrapup` method cannot be used reliably to free allocated memory. Instead, you should free memory from the previous run in the `setup` or `begin` method, prior to allocating new memory, and in the destructor.

public, protected, private

These three keywords allow the user to declare extra members for the class with the desired protection. The syntax is:

```
protkey { body }
```

where *`protkey`* is `public`, `protected`, or `private`. Example, from the `XMgraph` star:

```
protected {
      XGraph graph;
      double index;
}
```

This defines an instance of the class `XGraph`, defined in the Ptolemy kernel, and a double-precision number. If any of the added members require arguments for their constructors, use the `conscalls` item to specify them.

ccinclude, hinclude

These directives cause the `.cc` file, or the `.h` file, to `#include` extra files. A certain number of files are automatically included, when the preprocessor can determine that they are needed, so they do not need to be explicitly specified. The syntax is:

```
ccinclude { inclist }
hinclude { inclist }
```

where *`inclist`* is a comma-separated list of include files. Each filename must be surrounded either by quotation marks or by "<" and ">" (for system include files like `<math.h>`).

code

This keyword allows the user to specify a section of arbitrary C++ code. This code is inserted into the `.cc` file after the include files but before everything else; it can be used to define static non-class functions, declare external variables, or anything else. The outermost pair of curly braces is stripped. The syntax is:

```
code { body }
```

header

This keyword allows the user to specify an arbitrary set of definitions that will appear in the header file. Everything between the curly braces is inserted into the `.h` file after the include files but before everything else. This can be used, for example, to define classes used by your star. The outermost pair of curly braces is stripped.

method

The `method` item provides a fully general way to specify an additional method for the class of star that is being defined. Here is an example:

```
virtual method {
      name { exec }
      access { protected }
      arglist { "(const char* extraOpts)" }
      type { void }
      code {
           // code for the exec method goes here
      }
}
```

An optional function type specification may appear before the `method` keyword, which must be one of the following:

```
virtual
inline
pure
pure virtual
inline virtual
```

The `virtual` keyword makes a virtual member function. If the `pure virtual` keyword is given, a pure virtual member function is declared (there must be no `code` item in this case). The function type `pure` is a synonym for `pure virtual`. The `inline` function type declares the function to be inline.

Here are the `method` subitems:

`name`: The name of the method. This is a required item.

`access`: The level of access for the method, one of `public`, `protected`, or `private`. If the item is omitted, `protected` is assumed.

`arglist`: The argument list, including the outermost parentheses, for the method as a quoted string. If this is omitted, the method has no arguments.

`type`: The return type of the method. If the return type is not a single identifier, you must put quotes around it. If this is omitted, the return type is `void` (no value is returned).

`code`: The code that implements the method. This is a required item, unless the `pure` keyword appears, in which case this item *cannot* appear.

exectime

This item defines the optional `myExecTime()` function, which is used in code generation to specify how many time units are required to execute the star's code. The syntax is:

```
exectime { body }
```

The optional keyword `inline` may appear before the `exectime` keyword. The *`body`* defines the body of a function that returns an integer value.

codeblock

Codeblocks are parametrized blocks of code for use in code generation stars. Their use and format is discussed in detail in the code generation chapters. The syntax is:

```
codeblock {
      code
      ...
}
```

2.4 Writing C++ code for stars

This section assumes a knowledge of the C++ language; no attempt will be made to teach the language. We recommend "C++ Primer, Second Edition", by Stanley Lippman (from Addison-Wesley) for those new to the language. Chapter 3, "Infrastructure for Star Writers", is also highly recommended reading for those who will be writing stars, since it explains some of the more generic and useful classes defined in the Ptolemy kernel. Many of these are useful in stars.

C++ code segments are an important part of any star definition. They can appear in the `setup`, `begin`, `go`, `wrapup`, `constructor`, `destructor`, `exectime`, `header`, `code`, and `method` directives in the Ptolemy preprocessor. These directives all include a body of arbitrary C++ code, enclosed by curly braces, "`{`" and "`}`". In all but the `code` and `header` directives, the C++ code between braces defines the body of a method of the star class. Methods can access any member of the class, including portholes (for input and output), states, and members defined with the `public`, `protected`, and `private` directives.

2.4.1 The structure of a Ptolemy star

In general, the task of a Ptolemy star is to receive input particles and/or produce output particles; in addition, there may be side effects (reading or writing files, displaying graphs, or even updating shared data structures). As for all C++ objects, the constructor is called when the star is created, and the destructor is called when it is destroyed. In addition, the `setup` and `begin` methods, if any, are called every time a new simulation run is started, the `go` method (which always exists except for stars like `BlackHole` and `Null` that do nothing) is called each time a star is executed, and the `wrapup` method is called after the simulation run completes without errors.

2.4.2 Reading inputs and writing outputs

The precise mechanism for references to input and output portholes depends somewhat on the domain. This is because stars in the domain `XXX` use objects of class `InXXXPort` and `OutXXXPort` (derived from `PortHole`) for input and output, respectively. The examples we use here are for the SDF domain. See the appropriate domain chapter for variations that apply to other domains.

PortHoles and Particles

In the SDF domain, normal inputs and outputs become members of type `InSDFPort` and `OutSDFPort` after the preprocessor is finished. These are derived from base class `PortHole`. For example, given the following directive in the `defstar` of an SDF star,

```
input {
      name {in}
      type {float}
}
```

a member named `in`, of type `InSDFPort`, will become part of the star.

We are not usually interested in directly accessing these porthole classes, but rather wish to read or write data through the portholes. All data passing through a porthole is derived from base class `Particle`. Each particle contains data of the type specified in the `type` subdirective of the `input` or `output` directive.

The operator "`%`" operating on a porthole returns a reference to a particle. Consider the following example:

```
go {
      Particle& currentSample = in%0;
      Particle& pastSample = in%1;
      ...
}
```

The right-hand argument to the "`%`" operator specifies the delay of the access. A zero always means the most recent particle. A one means the particle arriving just before the most recent particle. The same rules apply to outputs. Given an output named `out`, the same particles that are read from `in` can be written to `out` in the same order as follows:

```
go {
      ...
      out%1 = pastSample;
      out%0 = currentSample;
}
```

This works because `out%n` returns a *reference* to a particle, and hence can accept an assignment. The assignment operator for the class `Particle` is overloaded to make a copy of the data field of the particle.

Operating directly on class `Particle`, as in the above examples, is useful for writing stars that accept `anytype` of input. The operations need not concern themselves with the type of data contained by the particle. But it is far more common to operate numerically on the data carried by a particle. This can be done using a cast to a compatible type. For example, since `in` above is of type `float`, its data can be accessed as follows:

```
go {
      Particle& currentSample = in%0;
      double value = double(currentSample);
      ...
}
```

or more concisely,

```
go {
      double value = double(in%0);
      ...
}
```

The expression `double(in%0)` can be used anywhere that a double can be used. In many contexts, where there is no ambiguity, the conversion operator can be omitted:

```
      double value = in%0;
```

However, since conversion operators are defined to convert particles to several types, it is often necessary to indicate precisely which type conversion is desired.

To write data to an output porthole, note that the right-hand side of the assignment operator should be of type `Particle`, as shown in the above example. An operator “<<” is defined for particle classes to make this more convenient. Consider the following example:

```
go {
      float t;
      t = some value to be sent to the output
      out%0 << t;
}
```

Note the distinction between the “<<” operator and the assignment operator; the latter operator copies Particles, the former operator loads data into particles. The type of the right-side operand of “<<” may be `int`, `float`, `double`, `Fix`, `Complex` or `Envelope`; the appropriate type conversion will be performed. For more information on the `Envelope` and `Message` types, please see the chapter “Data Types” on page 4-1.

SDF PortHole parameters

In the above example, where `in%1` was referenced, some special action is required to tell Ptolemy that past input particles are to be saved. Special action is also required to tell the SDF scheduler how many particles will be consumed at each input and produced at each output when a star fires. This information can be provided through a call to `setSDFParams` in the `setup` method. This has the syntax

```
setup {
      name.setSDFParams(multiplicity, past)
}
```

where *name* is the name of the input or output porthole, *multiplicity* is the number of particles consumed or produced, and *past* is the maximum value that *offset* can take in any expression of the form *name%offset*. For example, if the go method references *name*%0 and *name*%1, then *past* would have to be at least one. It is zero by default.

Multiple PortHoles

Sometimes a star should be defined with *n* input portholes or *n* output portholes, where *n* is variable. This is supported by the class MultiPortHole and its derived classes. An object of this class has a sequential list of PortHoles. For SDF, we have the specialized derived class MultiInSDFPort (which contains InSDFPorts) and MultiOutSDFPort (which contains OutSDFPorts).

Defining a multiple porthole is easy, as illustrated next:

```
defstar {
      ...
      inmulti {
            name {input_name}
            type {input_type}
      }
      outmulti {
            name {output_name}
            type {output_type}
      }
      ...
}
```

To successively access individual portholes in a MultiPortHole, the MPHIter iterator class should be used. Iterators are explained in more detail in “Iterators” on page 3-10. Consider the following code segment from the definition of the SDF Fork star:

```
input {
      name{input}
      type{ANYTYPE}
}
outmulti {
      name{output}
      type{= input}
}
go {
      MPHIter nextp(output);
      PortHole* p;
      while ((p = nextp++) != 0)
            (*p)%0 = input%0;
}
```

A single input porthole supplies a particle that gets copied to any number of output portholes. The `type` of the output `MultiPortHole` is inherited from the type of the input. The first line of the `go` method creates an `MPHIter` iterator called `nextp`, initialized to point to portholes in `output`. The "++" operator on the iterator returns a pointer to the next porthole in the list, until there are no more portholes, at which time it returns `NULL`. So the `while` construct steps through all output portholes, copying the input particle data to each one.

Consider another example, taken from the SDF `Add` star:

```
inmulti {
      name {input}
      type {float}
}
output {
      name {output}
      type {float}
}
go {
      MPHIter nexti(input);
      PortHole *p;
      double sum = 0.0;
      while ((p = nexti++) != 0)
            sum += double((*p)%0);
      output%0 << sum;
}
```

Again, an `MPHIter` iterator named `nexti` is created and used to access the inputs.

Upon occasion, the `numberPorts` method of class `MultiPortHole`, which returns the number of ports, is useful. This is called simply as *`portname`*`.numberPorts()`, and it returns an `int`.

Type conversion

The type conversion operators and “<<” operators are defined as virtual methods in the base class `Particle`. There are never really objects of class `Particle` in the system; instead, there are objects of class `IntParticle`, `FloatParticle`, `ComplexParticle`, and `FixParticle`, which hold data of type `int`, `double` (not float!), `Complex`, and `Fix`, respectively (there are also `MessageParticle` and a variety of matrix particles, described later). The conversion and loading operators are designed to “do the right thing” when an attempt is made to convert between mismatched types.

Clearly we can convert an `int` to a `double` or `Complex`, or a `double` to a `Complex`, with no loss of information. Attempts to convert in the opposite direction work as follows: conversion of a `Complex` to a `double` produces the magnitude of the complex number. Conversion of a `double` to an `int` produces the greatest integer that is less than or equal to the `double` value. There are also operators to convert to or from `float` and `Fix`.

Each particle also has a virtual `print` method, so a star that writes particles to a file can accept `anytype`.

2.4.3 States

A state is defined by the `state` directive. The star can use a state to store data values, remembering them from one invocation to another. They differ from ordinary members of the star, which are defined using the `public`, `protected`, and `private` directives, in that they have a name, and can be accessed from outside the star in systematic ways. For instance, the graphical interface `pigi` permits the user to set any state with the `A_SETTABLE` attribute to some value prior to a run, using the *edit-params* command. The interpreter provides similar functionality through the `setstate` command. The state attributes are set in the `state` directive. A state may be modified by the star code during a run. The attribute `A_NONCONSTANT` is used as a pragma to mark a state as one that gets modified during a run. There is currently no mechanism for checking the correctness of these attributes.

All states are derived from the base class `State`, defined in the Ptolemy kernel. The derived state classes currently defined in the kernel are `FloatState`, `IntState`, `ComplexState`, `StringState`, `FloatArrayState`, `IntArrayState`, `ComplexArrayState`, and `StringArrayState`.

A state can be used in a star method just like the corresponding predefined data types. As an example, suppose the star definition contains the following directive:

```
state {
      name { myState }
      type { float }
      default { 1.0 }
      descriptor { Gain parameter. }
}
```

This will define a member of class `FloatState` with default value 1.0. No attributes are defined, so `A_CONSTANT` and `A_SETTABLE`, the default attributes, are assumed. To use the value of a state, it should be cast to type `double`, either explicitly by the programmer or implicitly by the context. For example, the value of this state can be accessed in the `go` method as follows:

```
go {
     output%0 << double(myState) * double(input%0);
}
```

The references to input and output are explained above. The reference to `myState` has an explicit cast to `double`; this cast is defined in `FloatState` class. Similarly, a cast to `int` is available for `IntState`, to `Complex` from `ComplexState`, and to `const char*` for `Stringstate`). In principle, it is possible to rely on the compiler to automatically invoke this cast. However:

Warning: some compilers (notably some versions of g++) may not choose the expected cast. In particular, g++ has been known to cast everything to `Fix` if the explicit cast is omitted in expressions similar to that above. The arithmetic is then performed using fixed-point point computations. This will be dramatically slower than double or integer arithmetic, and may yield unexpected results. It is best to explicitly cast states to the desired form. An exception is with simple assignment statements, like

```
double stateValue = myName;
```

Even g++ gets this right. Explicit casting should be used whenever a state is used in an expression. For example, from the setup method of the `SDFChop` star, in which `use_past_inputs` is an integer state,

```
if ( int(use_past_inputs) )
  input.setSDFParams(int(nread),int(nread)+int(offset)-1);
else
  input.setSDFParams(int(nread),int(nread)-1);
```

Note that the type `Complex` is not a fundamental part of C++. We have implemented a subset of the `Complex` class as defined by several library vendors; we use our own version for maximum portability. Using the `ComplexState` class will automatically ensure the inclusion of the appropriate header files. A member of the `Complex` class can be initialized and operated upon any number of ways. For details, see “The Complex data type” on page 4-1.

A state may be updated by ordinary assignment in C++, as in the following lines

```
double t = expression;
myState = t;
```

This works because the `FloatState` class definition has overloaded the assignment operator (“=”) to set its value from a `double`. Similarly, an `IntState` can be set from an `int` and a `StringState` can be set from a `char*` or `const char*` .

2.4.4 Array States

The `ArrayState` classes (`FloatArrayState`, `IntArrayState` and `ComplexArrayState`) are used to store arrays of data. For example,

```
state {
      name { taps }
      type { FloatArray }
      default { "0.0 0.0 0.0 0.0" }
      descriptor { An array of length four. }
}
```

defines an array of type `double` with dimension four, with each element initialized to zero. Quotes must surround the initial values. Alternatively, you can specify a file name with a prefix <. If you have a file named `foo` that contains the default values for an array state, you can write,

```
default { "< foo" }
```

If you expect others to be able to use your star, however, you should specify the default filename using a full path. For instance,

```
default { "< ~/user_name/directory/foo" }
```

For default files installed in the Ptolemy directory tree, this should read:

```
default { "< $PTOLEMY/directory/foo" }
```

The format of the file is also a sequence of data separated by spaces (or newlines, tabs, or commas). File input can be combined with direct data input as in

```
default { "< foo 2.0" }
default { "0.5 < foo < bar" }
```

A “repeat” notation is also supported for `ArrayState` objects: the two value strings

```
default { "1.0 [5]" }
default { "1.0 1.0 1.0 1.0 1.0" }
```

are equivalent. Any integer expression may appear inside the brackets `[]`. The number of elements in an `ArrayState` can be determined by calling its `size` method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an `ArrayState`, suppose `fState` is a `FloatState` and `aState` is a `FloatArrayState`. The accesses, like those in the following lines, are routine:

```
fState = aState[1] + 0.5;
aState[1] = (double)fState * 10.0;
aState[0] = (double)fState * aState[2];
```

For a more complete example of the use of `FloatArrayState`, consider the `FIR` star defined below. Note that this is a simplified version of the SDF `FIR` star that does not permit interpolation or decimation.

```
defstar {
      name {FIR}
      domain {SDF}
      desc {
A Finite Impulse Response (FIR) filter.
      }
      input {
            name {signalIn}
            type {float}
      }
      output {
            name {signalOut}
            type {float}
      }
      state {
            name {taps}
            type {floatarray}
            default { "-.04 -.001 .17 .37 .37 .17 -.0018 -.04" }
            desc { Filter tap values. }
      }
      setup {
            // tell the PortHole the maximum delay we will use
            signalIn.setSDFParams(1, taps.size() - 1);
      }
      go {
            double out = 0.0;
            for (int i = 0; i < taps.size(); i++)
                  out += taps[i] * double(signalIn%i);
            signalOut%0 << out;
      }
}
```

Notice the `setup` method; this is necessary to allocate a buffer in the input `PortHole` large enough to hold the particles that are accessed in the `go` method. Notice the use of the `size` method of the `FloatArrayState`.

We now illustrate an `ptcl` interpreter session using the above `FIR` star. Assume there is a galaxy called `singen` that generates a sine wave. you can use it with the `FIR` star, as in:

```
star foop singen
star fir FIR
star printer Printer
connect foop output fir signalIn
connect fir signalOut printer input
print fir
Star: mainGalaxy.fir
      ...
States in the star fir:
mainGalaxy.fir.taps type: FloatArray
initial value: -.040609 -.001628 .17853 .37665 .37665 .17853
-.001628 -.040609
current value:
0 -0.040609
1 -0.001628
```

```
2 .17853
3 .37665
4 .37665
5 .17853
6 −0.001628
7 −0.040609
```

Then you can redefine taps by reading them from a file “`foo`”, which contains the data:

```
1.1
−2.2
3.3
−4.4
```

The resulting interpreter commands are:

```
setstate fir taps "<foo 5.5"
print fir
Star: mainGalaxy.fir
      ...
States in the star fir:
mainGalaxy.fir.taps type: FloatArray
initial value: <foo 5.5
current value:
0 1.1
1 −2.2
2 3.3
3 −4.4
4 5.5
PTOLEMY:
```

This illustrates that *both* the contents and the size of a `FloatArrayState` are changed by a `setstate` command. Also, notice that file values may be combined with string values; when

```
< filename
```

occurs in an *initial value*, it is processed exactly as if the whole file were substituted at that point.

2.5 Modifying PortHoles and States in Derived Classes

When one star is derived from another, it inherits all the states of the base class star. Sometimes we want to modify some aspect of the behavior of a base class state in the derived class. This is done by placing calls to member functions of the state in the constructor of the derived star. Useful functions include `setInitValue` to change the default value, and `setAttibututes` and `clearAttributes` to modify attributes.

When creating new stars derived from stars already in the system, you will often also wish to customize them by adding new ports or states. In addition, you may wish to remove ports or states. Although, strictly speaking, you cannot do this, you can achieve the desired effect by simply hiding them from the user.

The following code will hide a particular state named `statename` from the user:

```
constructor {
      statename.clearAttributes(A_SETTABLE);
}
```

This means that when the user invokes "edit-params" in `pigi`, statename will not appear as one of the parameters of the star. Of course, the state can still be set and used within the code defining the star.

The same effect can be achieved with outputs or inputs. For instance, given an output named `output`, you can use the following code:

```
constructor {
      output.setAttributes(P_HIDDEN);
}
```

This means that when you create an icon for this star, no terminal will appear for this port. This is most useful when `output` is a multiporthole, because this means simply that there will be zero instances of the individual portholes.

This technique can also be used to hide individual portholes, however, the porthole will still be present, so it must be used with caution. Most domains do not allow disconnected portholes, and will flag an error. You can explicitly connect the port within the body of the star (see the kernel manual).

2.6 Programming examples

The following star has no inputs, just an output. The source star generates a linearly increasing or decreasing sequence of float particles on its output. The state *`value`* is initialized to define the value of the first *`output`*. Each time the star `go` method fires, the *`value`* state is updated to store the next *`output`* value. Hence, the attributes of the *`value`* state are set so that the state can be overwritten by the star's methods. By default, the star will generate the output sequence 0.0, 1.0, 2.0, etc.

```
defstar {
      name { Ramp }
      domain { SDF }
      desc {
Generates a ramp signal, starting at "value" (default 0)
with step size "step" (default 1).
      }
      output {
            name { output }
            type { float }
      }
      state {
            name { step }
            type { float }
            default { 1.0 }
            desc { Increment from one sample to the next. }
```

```
        }
        state {
              name { value }
              type { float }
              default { 0.0 }
              desc { Initial (or latest) value output by Ramp. }
              attributes { A_SETTABLE|A_NONCONSTANT }
        }
        go {
              double t = double(value);
              output%0 << t;
              t += step;
              value = t;
        }
  }
```

The next example is the `Gain` star, which multiplies its input by a constant and outputs the result:

```
  defstar {
        name { Gain }
        domain { SDF }
        desc { Amplifier: output is input times "gain" (default 1.0). }
        input {
              name { input }
              type { float }
        }
        output {
              name { output }
              type { float }
        }
        state {
              name { gain }
              type { float }
              default { "1.0" }
              desc { Gain of the star. }
        }
        go {
              output%0 << double(gain) * double(input%0);
        }
  }
```

The following example of the `Printer` star illustrates multiple inputs, `ANYTYPE` inputs, and the use of the `print` method of the `Particle` class.

```
  defstar {
        name { Printer }
        domain { SDF }
        inmulti {
              name { input }
              type { ANYTYPE }
        }
        state {
```

```
                name { fileName }
                type { string }
                default { "<cout>" }
                desc { Filename for output. }
        }
        hinclude { "pt_fstream.h" }
        protected {
                pt_ofstream *p_out;
        }
        constructor { p_out = 0;}
        destructor { LOG_DEL; delete p_out;}
        setup {
                delete p_out;
                p_out = new pt_ofstream(fileName);
        }
        go {
                pt_ofstream& output = *p_out;
                MPHIter nexti(input);
                PortHole* p;
                while ((p = nexti++) != 0)
                        output << ((*p)%0).print() << "\t";
                output << "\n";
        }
}
```

This star is *polymorphic* since it can operate on any type of input. Note that the default value of the output filename is `<cout>`, which causes the output to go to the standard output. This and other aspects of the `pt_ofstream` output stream class are explained below in "Extended input and output stream classes" on page 3-2. The iterator `nexti` used to scan the input is explained in "Iterators" on page 3-10.

2.7 Preventing Memory Leaks in C++ Code

Memory leaks occur when new memory is allocated dynamically and never deallocated. In C programs, new memory is allocated by the `malloc` or `calloc` functions, and deallocated by the `free` function. In C++, new memory is usually allocated by the `new` operator and deallocated by the `delete` or the `delete []` operator. The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program. We have taken extensive steps to eliminate memory leaks in the Ptolemy software environment by following the guidelines below and by tracking memory leaks with Purify (a commercial tool from Pure Software Inc.).

One of the most common mistakes leading to memory leaks is applying the wrong `delete` operator. The `delete` operator should be used to free a single allocated class or data value, whereas the `delete []` operator should be used to free an array of data values. In C programming, the `free` function does not make this distinction.

Another common mistake is overwriting a variable containing dynamic memory without freeing any existing memory first. For example, assume that `thestring` is a data member of a class, and in one of the methods (other than the constructor), there is the following statement:

```
thestring = new char[buflen];
```

This code should be

```
delete [] thestring;
thestring = new char[buflen];
```

Using `delete` is not necessary in a class's constructor because the data member would not have been allocated previously.

In writing Ptolemy stars, the `delete` operator should be applied to variables containing dynamic memory in both the star's setup and destructor methods. In the star's constructor method, the variables containing dynamic memory should be initialized to zero. By freeing memory in both the setup and destructor methods, one covers all possible cases of memory leaks during simulation. Deallocating memory in the setup method handles the case in which the user restarts a simulation, whereas deallocating memory in the destructor covers the case in which the user exits a simulation. This includes the cases that arise when error messages are generated. For an example implementation, see the implementation of the `SDFPrinter` star given in Section 2.6.

Another common mistake is not paying attention to the kinds of strings returned by functions. The function `savestring` returns a new string dynamically allocated and should be deleted when no longer used. The `expandPathName`, `tempFileName`, and `makeLower` functions return new strings, as does the `Target::writeFileName` method. Therefore, the strings returned by these routines should be deleted when they are no longer needed, and code such as

```
savestring( expandPathName(s) )
```

is redundant and should be simplified to

```
expandPathName(s)
```

to avoid a memory leak due to not keeping track of the dynamic memory returned by the function `savestring`.

Occasionally, dynamic memory is being used when instead local memory could have been used. For example, if a variable is only used as a local variable inside a method or function and the value of the local variable is not returned or passed to outside the method or function, then it would be better to simply use local memory. For example,

```
char* localstring = new char[len + 1];
if ( person == absent ) return;
strcpy(localstring, otherstring);
delete [] localstring;
return;
```

could easily return without deallocating `localstring`. The code should be rewritten to use either the `StringList` or `InfString` class, e.g.,

```
InfString localstring;
if ( person == absent ) return;
localstring = otherstring;
return;
```

Both `StringList` and `InfString` can manage the construction of strings of arbitrary size. When a function or method exits, the destructors of the `StringList` and `Inf-`

`String` variables will automatically be called which will deallocate their memory. Casts have been defined that will convert `StringList` to a `const char*` string and `InfString` to a `const char*` or a `char*` string, so that instances of the `StringList` and `InfString` classes can be passed as is into routines that take character array (string) arguments. A good example of using the `StringList` class is in the function `compile` in the file `$PTOLEMY/src/pigilib/pigiLoader.cc`. A simpler example from the same file is the `noPermission` function which builds up an error message into a single string:

```
StringList sl = msg;
sl << file << ": " << sys_errlist[errno];
ErrAdd(sl);
```

The `errAdd` function takes a `const char*` argument, so `sl` will converted automatically to a `const char*` string by the C++ compiler.

Instead of using the new and delete operators, it is tempting to use constructs like

```
char localstring[buflen + 1];
```

in which `buflen` is a variable, because the compiler will automatically handle the deallocation of the memory. Unfortunately, this syntax is a Gnu extension and not portable to other C++ compilers. Instead, the `StringList` and `InfString` classes should be used, as the previous example involving `localstring` illustrates.

Sometimes the return value from a routine that returns dynamic memory is not stored, and therefore, the pointer to the dynamic memory gets lost. This occurs, for example, in nested function calls. Code such as

```
puts( savestring(s) );
```

should be written as

```
const char* newstring = savestring(s);
puts( newstring );
delete [] newstring;
```

Several places in Ptolemy, especially in the schedulers and targets, rely on the `hashstring` function, which returns dynamic memory. This dynamic memory, however, should *not* be deallocated because it may be reused by other calls to `hashstring`. It is the responsibility of the `hashstring` function to deallocate any memory it has allocated.