Parameter Lists

Neil N. Carlson

July 2013
Version 1.0

Abstract

A parameter list is a hierarchical data structure consisting of a collection of parameter name/value pairs. A value can be a scalar or array of arbitrary type, or a parameter list itself. Parameter lists are intended as a convenient way to pass information between different software layers. The module `parameter_list_type` implements the data structure and basic methods for working with it, and the module `parameter_list_json` provides additional procedures for parameter list input/output using JSON-format text.

1 Synopsis

Usage

```
use parameter_list_type
use parameter_list_json
```

Derived Types

`parameter_list`, `parameter_list_iterator`, `parameter_entry`, `any_scalar`, `any_vector`

Procedures

`parameter_list_from_json_stream`, `parameter_list_to_json`

2 Introduction

A parameter list is a hierarchical data structure consisting of a collection of parameter name/value pairs. A value can be a scalar or array of arbitrary type, or a parameter list itself. In the latter case we regard the parameter as a sublist parameter. Parameter lists are used to pass information between different software layers. Long argument lists can be shortened by gathering some of the arguments into a parameter list and passing it instead. Sublists can be used for arguments intended to be passed in lower-level calls, and so forth. While a parameter list can be populated through a sequence of calls to define its parameters, it can also be defined using JSON text from an input file, and therein lies its real power. If

JSON is used to express text input, parameter lists provide a powerful and flexible means for easily disseminating that input throughout the entire software stack.

Parameter lists are intended to pass lightweight data: scalars or (very) small arrays, typically of intrinsic primitive types. Values of arbitrary derived type can be used but great caution must be used when doing so because the get/set methods make shallow copies of the value; see *Limitation on values* below. Also such values cannot be input/output using JSON-format text.

This implementation is inspired by, and modeled after, the `Teuchos::ParameterList` C++ class from Trilinos (`http://trilinos.sandia.gov`).

3 The parameter_list derived type

The derived type `parameter_list` implements the parameter list data structure. It has the following properties.

- Assignment is defined for `parameter_list` variables with the expected semantics. The lhs parameter list is first deleted, and then defined with the same parameters and values as the rhs parameter list, becoming an independent copy of the rhs parameter list; but see *Limitation on values* below.
- The structure constructor `parameter_list()` evaluates to an empty parameter list, and `parameter_list` variables come into existence as empty parameter lists.
- `Parameter_list` objects are properly finalized when they are deallocated or otherwise cease to exist.

Values of the class `parameter_entry` are held as the parameter values in a parameter list. There are three different derived types of this class: `any_scalar`, which stores a scalar value of any intrinsic or derived type; `any_vector`, which stores a rank-1 array value of any intrinsic or derived type; and `parameter_list` itself, which gives rise to the hierarchical character of the parameter list data structure. For the most commo

any_scalar stores a scalar value of any intrinsic or derived type;

any_vector stores a rank-1 array value of any intrinsic or derived type;

parameter_list

The derived type has the following type bound procedures. Some have the optional intent-out arguments `stat` and `errmsg`. If the integer `stat` is present, it is assigned the value 0 if no error was encountered; otherwise it is assigned a non-zero value. In the latter case, the allocatable deferred-length character string `errmsg`, if present, is assigned an explanatory message. If `stat` is not present and an error occurs, the error message is written to the preconnected error unit and the program terminates.

3.1 Type bound subroutines

`set(name, value [,stat [,errmsg]])`
: defines a parameter with the given `name` and assigns it the given `value`, which may be a scalar or rank-1 array of any type. A copy of the passed value, as created by sourced allocation, is stored in the parameter list and thus caution should be exercised with derived type values; see *Limitations on values* below. If the parameter already exists, it must not be a sublist parameter and its existing value must have the same rank as `value`, but not necessarily the same type; its value is overwritten with `value`.

`get(name, value [,default] [,stat [,errmsg]])`
: retrieves the value of the parameter `name`. A copy of the value is returned in `value`, which may be a scalar or rank-1 array of the following intrinsic types: `integer(int32)`, `integer(int64)`, `real(real32)`, `real(real64)`, default `logical`, and default `character`. The kind parameters are those from the intrinsic module `iso_fortran_env`, and should cover the default integer and real kinds, as well as double precision. An array `value` must be allocatable and a character `value` must be deferred-length allocatable. In these latter cases, `value` is allocated with the proper size/length to hold the parameter value. If present, the optional argument `default` must have the same type, kind, and rank as `value`. If the named parameter does not exist, it is created with the value prescribed by `default`, and that value is return in `value`. It is an error if the named parameter does not exist and `default` is not present. It is an error if the named parameter is a sublist. It is an error if the type, kind, and rank of `value` does not match the stored value of the named parameter. Use `get_any` when the type of the parameter value is not one of those handled by this method.

`get_any(name, value [,default] [,stat [,errmsg]])`
: retrieves the value of the parameter `name`. A copy of the value is returned in `value`, which is an allocatable `class(*)` variable or rank-1 array. This is a more general version of `get` that can retrieve any type of parameter value. The downside of `get_any` is that the application code must use a select-type construct in order to use the returned value, making it more complex to use. If present, the optional argument `default` must have the same rank as `value`. If the named parameter does not exist, it is created with the value prescribed by `default`, and that value is returned in `value`. It is an error if the named parameter does not exist and `default` is not present. It is an error if the named parameter is a sublist. It is an error if the rank of `value` does not match that of the stored value of the named parameter.

  N.B. It would have been desirable to merge `get` and `get_any` into a single generic subroutine, but this isn't possible because the specific subroutines have an ambiguous interface—the `class(*)` value argument of `get_any` also matches any of the specific types of the value argument of `get`.

3.2 Type bound functions

`sublist(name [,stat [,errmsg]])`
: returns a `type(parameter_list)` pointer to the named parameter sublist. The pa-

rameter is created with an empty sublist value if it does not already exist. It is an error if the parameter exists but is not a sublist.

`is_parameter(name)`
: returns true if there is a parameter with the given `name`; otherwise it returns false.

`is_sublist(name)`
: returns true if there is a sublist parameter with the given `name`; otherwise it returns false.

`count()`
: returns the number of parameters stored in the parameter list.

3.3 Limitation on values

This implementation uses the `map_any` derived type to store the parameter name/value pairs. The values stored in that data structure are copies of the values passed to the `set` method, but they are shallow copies as created by sourced allocation. For intrinsic types these are genuine copies. For derived type values, the literal contents of the object are copied, which for a pointer component means that a copy of the pointer is made but not a copy of the target; the original pointer and its copy will have the same target. This also applies to parameter list assignment. Thus great caution should be used when using values of derived types that contain pointer components, either directly or indirectly.

4 The parameter_list_iterator derived type

Parameter values can be accessed in a parameter list directly, but only if the parameter names are known. The derived type `parameter_list_iterator` provides a means of iterating through the parameters in a `parameter_list` object; that is, sequentially visiting each parameter in the list once and only once. Once initialized, a `parameter_list_iterator` object is positioned at a particular parameter of its associated parameter list, or at a pseudo-position *the end*, and can be queried for the name and value of that parameter.

Scalar assignment is defined for `parameter_list_iterator` objects. The lhs iterator becomes associated with the same parameter list as the rhs iterator and is positioned at the same parameter. Subsequent changes to one iterator do not affect the other.

An iterator is initialized via assignment, normally using a rhs that is a structure constructor expression (see below).

4.1 Constructors

`parameter_list_iterator(plist [,sublists_only])`
: evaluates to an iterator positioned at the initial parameter the parameter list `plist`, or the end if no parameters exist. If the optional argument `sublists_only` is present with value true, parameters other than sublists are skipped by the iterator.

4.2 Type bound subroutines

`next()`
: advances the iterator to the next parameter in the list, or to the end if there are no more parameters remaining to be visited. This call has no effect if the iterator is already positioned at the end.

4.3 Type bound functions

`at_end()`
: returns true if the iterator is positioned and the end; otherwise it returns false.

`name()`
: returns the name of the current parameter. The iterator must not be positioned at the end.

`value()`
: returns a `class(parameter_entry)` pointer to an object that holds the value of the current parameter. The iterator must not be positioned at the end. The target of the pointer will be of one of three dynamic types:

  `parameter_list` if the parameter value is a sublist;

  `any_scalar` if the parameter has a scalar value;

  `any_vector` if the parameter has a vector value.

  A select-type construct with stanzas for these three types is required to access the value. For sublists it is easier to use the `is_list` method to identify whether the current parameter is a sublist, and if so use the `sublist` method to acess the sublist.

`is_list()`
: returns true if the current parameter value is a sublist; otherwise it returns false. The iterator must not be positioned at the end.

`is_scalar()`
: returns true if the current parameter has a scalar value; otherwise it returns false. The iterator must not be positioned at the end.

`is_vector()`
: returns true if the current parameter has a vector value; otherwise it returns false. The iterator must not be positioned at the end.

`sublist()`
: returns a `type(parameter_list)` pointer that is associated with the current parameter value if it is a sublist; otherwise it returns a `null()` pointer.

`count()`
: returns the number of remaining parameters, including the current one.

5 Parameter list input/output using JSON

JSON is a widely-used data format (`http://www.json.org`). It is lightweight, flexible, and easy for humans to read and write—characteristics that make it an ideal format for data interchange. A parameter list whose values are of primitive intrinsic types can be represented quite naturally as JSON text that conforms to a subset of the JSON format:

- A parameter list is represented by a JSON *object*, which is an unordered list of comma-separated *name*`:`*value* pairs enclosed in braces (`{` and `}`).
- A parameter name and value are represented by a *name*`:`*value* pair of the object:
  - A *name* is a string enclosed in double quotes.
  - A *value* may be a string (in double quotes), an integer, a real number, or a logical value (literally `true` or `false`, not the Fortran constants).
  - A *value* may be also be a JSON *array*, which is an ordered list of comma-separated *values* enclosed in brackets (`[` and `]`). To represent an array parameter value, the values in a JSON array are limited to scalars of the same primitive type. JSON would allow array values of different types and even general JSON values (object and array).
  - A *value* may also be a JSON object which is interpreted as a parameter sublist.
  - Null values (`null`) are not allowed.
  - 0-sized arrays are not allowed.
- Comments (starting from `//` to the end of the line) are allowed; this is an extention to the JSON standard that is provided by the YAJL library that performs the actual parsing of the JSON text.

The `parameter_list_json` module provides the following procedures for creating a parameter list object from JSON text and for producing a JSON text representation of a parameter list object.

`call parameter_list_from_json_stream (unit, plist, errmsg)`
: reads JSON text from the logical unit `unit`, which must be connected for unformatted stream access, and creates the corresponding parameter list, as described above. The intent-out `type(parameter_list)` pointer argument `plist` returns the created parameter list. An unassociated return value indicates an error condition, in which case the allocatable deferred-length character argument `errmsg` is assigned an explanatory error message.

`call parameter_list_to_json (plist, unit)`
: writes the JSON text representation of the parameter list `plist` to `unit`, which must be connected for formatted write access. The parameter list values other than sublists must be of intrinsic primitive types that are representable in JSON: logical, integer, real, character.

6 Example

```
%%% SOME EXAMPLE CODE
```

7 Bugs

Bug reports and improvement suggestions should be directed to `neil.n.carlson@gmail.com`