ORNL/TM-2021/2041

Flibcpp User Manual

Seth R. Johnson

Sep. 2021

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT–BATTELLE, LLC, FOR THE U.S. DEPARTMENT OF ENERGY

ORNL/TM-2021/2041

Computational Sciences and Engineering Division

FLIBCPP USER MANUAL

Seth R. Johnson

Date Published: Sep. 2021

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

Flibcpp uses SWIG-Fortran to generate native Fortran-2003 interfaces to efficient and robust algorithms and data containers implemented in the C++ standard library.

1. INTRODUCTION

The Fortran programming language includes many mathematical functions but few of the generic, high-performance software algorithms and data containers needed by essentially all modern scientific software. Most Fortran software contains hand-written code for such algorithms that may be burdensome to maintain and inflexible, as well as unperformant and erroneous under certain conditions.

Flibcpp[1] is a library for use by application developers that provides native Fortran interfaces to existing high-quality algorithms and containers implemented in C++ and available on all modern computer systems.

Flibcpp defines a carefully crafted set of interface files written for the SWIG-Fortran code generator [JPE20], an extension of SWIG [Bea03]. These Fortran interfaces generate native Fortran proxy code that comprises a set of thin wrappers to selected functionality in the C++ standard library. The resulting code is a set of Fortran modules and C++ wrappers that expose a concise and well-defined interface that may be built into and distributed with the application.

The generated modules include functionality for efficient generic sorting and searching, set operations, random number generation, value mapping, string manipulation, and dynamically resizing vectors.

[1] This documentation is generated from Flibcpp v1.0.1+4846e7d.

2. INFRASTRUCTURE

Flibcpp is built using modern CMake, and it has no external dependencies. This makes installation and usage quite simple as long as you have a relatively recent software stack with a Fortran and compatible C++ compiler.

2.1 INSTALLATION

1. Download and install CMake if it's not already on your system. It is highly recommended to use a package manager such as Homebrew for Mac or YUM for Red Hat Linux.
2. Download the Flibcpp source code from GitHub if you haven't already.
3. Create a new build directory (for example purposes, create a subdirectory named `build` inside your downloaded source directory) and `cd` to it.
4. Run CMake: `cmake ..`
5. Make and install (by default it will install to `/usr/local`): `make install`.

By default, Flibcpp builds shared libraries. Add the CMake argument `-DBUILD_SHARED_LIBS:BOOL=OFF` to build static libraries.

2.2 DOWNSTREAM USAGE AS A LIBRARY

The Flibcpp library is most easily used when your downstream app is built with CMake. It should require a single line to initialize:

```
find_package(Flibcpp REQUIRED CONFIG)
```

and a single line to link against your app or library:

```
target_link_libraries(example_backend Flibcpp::flc_random Flibcpp::flc_algorithm)
```

If your installation prefix for Flibcpp is a system default path (such as `/usr/local`) or in your `$CMAKE_PREFIX_PATH` environment variable, it should automatically find the necessary CMake configure file.

An example Fortran application that depends only on Flibcpp is available on Github.

If you're using a simple standalone Makefile to build your Fortran code, you will have to inform the compiler of the proper include path, library path, and library names. Depending on your system configuration, you might have to also explicitly link your app against the compiler's C++ standard libraries using `-lstdc++`.

2.3 DOWNSTREAM USAGE AS A COMPONENT

Flibcpp's SWIG interface files can be used with your Fortran-accessible C++ project to seamlessly integrate the Flibcpp Fortran wrapper code with yours. To start, you must have the latest version of the SWIG+Fortran tool installed on your machine: the version of SWIG used by your installation of Flibcpp *must* match the version used by your downstream library/app. When you build Flibcpp for downstream SWIG usage, you must configure using `cmake -DFLIBCPP_USE_SWIG=ON ...`. This will cause the SWIG interface files to be installed to `${CMAKE_PREFIX_PATH}/include` to make them available downstream. Finally, in

your downstream SWIG interface code, instead of calling `%import <flc.i>` you must use `%include <import_flc.i>`. This is necessary to inject function declarations and other internal macros into your wrapper code.

At that point, all you have to do is (for example) `%import <flc_vector>` to allow `std::vector<double>` in your library headers to be wrapped by Flibcpp's `VectorReal8` Fortran proxy derived type.

An example C++/Fortran library that integrates with Flibcpp will be made available on Github.

2.4 DEVELOPING

If you are interested in extending the capabilities of Flibcpp, you will need the latest version of the SWIG+Fortran tool installed on your machine. When configuring CMake, you will want to configure using `cmake -DFLIBCPP_DEV=ON ..` to enable tests and documentation. Tests, examples, and documentation can be independently enabled using the `FLIBCPP_BUILD_TESTS`, `FLIBCPP_BUILD_EXAMPLES`, and `FLIBCPP_BUILD_DOCS` options.

3. CONVENTIONS

Since the C++ and Fortran binding code are generated by SWIG-Fortran, most conventions are based on its defaults and built-in wrappers as described in *the SWIG+Fortran manual page*. This section attempts to describe all conventions used by Flibcpp that may be ambiguous to either a C++ or Fortran user. Since Flibcpp's target audience is Fortran users, the library tries to follows existing Fortran conventions, and this document attempts to remain true to the official Fortran nomenclature as described in the ISO standards.

3.1 BASIC TYPES

The C++ standard library features many numeric classes and functions that take a template parameter corresponding to the type of a single element used by the class or algorithm. All such classes or functions that Flibcpp includes are templated on:

- 32-bit integers (`integer(4)` or `integer(C_INT32_T)`), with a suffix of `Int4` where applicable;
- 64-bit integers (`integer(8)` or `integer(C_INT64_T)`), with a suffix of `Int8` where applicable; and
- 64-bit reals (`real(8)` or `real(C_DOUBLE)`), with a suffix of `Real8` where applicable.

In general, most templated C++ functions use Fortran generic procedures to allow a single procedure name to operate on multiple types, and only the derived types (such as `VectorInt4`) require the suffix.

3.2 ERROR HANDLING

Some modules support error handling for checking input values. In all cases, the error status and a message can be accessed and cleared through the main `flc` module:

```
use flc, only : ierr, get_serr

! <snip>
if (ierr /= 0) then
  write(1,*) "Error", ierr, ":", get_serr()
  ! Reset the error flag to indicate that the error has been successfully
  ! handled.
  ierr = 0
fi
```

Since errors do not immediately exit, it is up to the application code to check for and clear them. Failing to clear the error code may cause the application to exit on a subsequent Flibcpp procedure call.

3.3 INDEXING

C and C++ use the convention that 0 corresponds to the first element in an array: specifically, it indicates an *offset* of zero from the array's beginning. In Fortran, the convention is for the first element to have an index of 1, so Flibcpp has the same convention.

This convention makes native array integration straightforward. For example, when the binary_search algorithm is used with any value `val` in the sorted array `arr`, the following logical expression will be true:

```
arr(binary_search(arr, val)) == val
```

An additional convention Flibcpp uses is for an invalid index to have a value of zero. Thus, the binary_search algorithm returns zero if asked to find an element that's not present.

3.4 DERIVED TYPE BEHAVIOR

The derived types defined by Flibcpp are all "proxy" objects that operate on *pointers* to C++-owned objects. Some derived type values (i.e. class instances) will "own" the associated data, while some will merely reference data owned by other C++ objects.

Note: Memory management in SWIG-Fortran-generated code is unintuitive and could change. If you have feedback, please contact the author.

3.4.1 CONSTRUCTION

Although Fortran separates the processes of allocation and initialization, C++ combines them into the single act of *construction*. Thus the proxy objects in Flibcpp can either be in an unassigned, deallocated state or in an allocated and internally consistent state.

Each derived type, such as the String type, is constructed using a *module procedure interface* with the same name as the type. This procedure, defined in the same module as the derived type, is a `function` that returns a "constructed" type:

```
use flc_string, only : String
type(String) :: s

s = String()
```

Most classes have more than one constructor (i.e. procedure interface) that gives the object a more substantial initial state. For example, numeric vectors can be constructed directly from a Fortran array:

```
use flc_vector, only : Vector => VectorInt4
type(Vector) :: v, v2
v = Vector([1, 2, 3, 5, 8, 13])
```

3.4.2 ASSIGNMENT

SWIG-Fortran, and consequently Flibcpp, defines assignment operators for its types that control memory ownership. Assignment for these types can be grouped into two categories: assignment directly from a Flibcpp function return value, and assignment from an existing Flibcpp derived type value.

Flibcpp (or any other SWIG-Fortran) wrapper code sets the correct ownership flag on a return value: non-owning for raw pointers and references; owning for return-by-value. When the left-hand side of an assignment is uninitialized, it captures the returned value and obtains the correct ownership flag. If the left-hand side *is* initialized, it is automatically destroyed first.

Assignment from within Fortran is like pointer assignment. The left-hand side becomes a non-owning reference to the right-hand side.

3.4.3 DESTRUCTION

Unlike native allocatable Fortran types, Flibcpp derived types are not automatically deallocated when ending a procedure. Therefore to avoid leaking memory, these derived type values must be explicitly cleaned up and released. This is done by the type-bound subroutine named `release`:

```
type(Vector), intent(inout) :: v
call v%release()
```

If the value `v` above owns the associated memory (i.e. if it was constructed in user code), then Flibcpp cleans up and deallocates the C++ instance, and sets `v` to an uninitialized state. If `v` merely points to existing C++ data, i.e. if it was assigned to the return result of a C++ accessor, then Flibcpp will simply set `v` to an uninitialized state.

4. MODULES

Flibcpp is organized into distinct modules whose structure mirrors the C++ standard library include headers.

The modules themselves are namespaced with a `flc_` prefix, so for example the `std::sort` algorithm, available in the <algorithm> header, can be obtained via:

```
use flc_algorithm, only : sort
```

4.1 ALGORITHM

The `flc_algorithm` module wraps C++ standard algorithm routines. Instead of taking pairs of iterators, the Flibcpp algorithm subroutines accept target-qualified one-dimensional arrays. All algorithms follow the indexing convention that the first element of an array has index 1, and an index of 0 indicates "not found".

4.1.1 SORTING

Sorting algorithms for numeric types default to increasing order when provided with a single array argument. Numeric sorting routines accept an optional second argument, a comparator function, which should return `true` if the first argument is strictly less than the right-hand side.

> **Warning:** For every value of `a` and `b`, the comparator `cmp` *must* satisfy `.not. (cmp(a, b) .and. cmp(b, a))`. If this strict ordering is not satisfied, some of the algorithms below may crash the program.

All sorting algorithms are *also* instantiated so that they accept an array of `type(C_PTR)` and a generic comparator function. **This enables arrays of any native Fortran object to be sorted**. See the generic sorting example for a demonstration.

4.1.1.1 sort

Sorting and checking order is a single simple subroutine call:

```
use flc_algorithm, only : sort
implicit none
integer, dimension(5) :: iarr = [ 2, 5, -2, 3, -10000]

call sort(iarr)
```

4.1.1.2 is_sorted

Checking the ordering of array is just as simple:

```
use flc_algorithm, only : is_sorted
integer, dimension(5) :: iarr = [ 2, 5, -2, 3, -10000]
logical :: sortitude

sortitude = is_sorted(iarr)
```

4.1.1.3 argsort

A routine that provides the indices that correspond to a sorted array, like Numpy's argsort , takes an array to analyze and an empty array of integers to fill:

```
use flc_algorithm, only : argsort, INDEX_INT
implicit none
integer, dimension(5) :: iarr = [ 2, 5, -2, 3, -10000]
integer(INDEX_INT), dimension(size(iarr)) :: idx

call argsort(iarr, idx)
write(*,*) iarr(idx) ! Prints the sorted array
```

Note that the index array is always a `INDEX_INT`, which is an alias to `C_INT`. On some compilers and platforms, this may be the same as native Fortran integer, but it's not guaranteed.

The `data` and `idx` arguments to `argsort` *must* be the same size. If the index array is larger than the data, invalid entries will be filled with zero.

4.1.2 SEARCHING

Like the sorting algorithms, searching algorithms are instantiated on numeric types and the C pointer type, and they provide an optional procedure pointer argument that allows the arrays to be ordered with an arbitrary comparator.

4.1.2.1 binary_search

A binary search can be performed on sorted data to efficiently find an element in a range. If the element is not found, the function returns zero; otherwise, it returns the Fortran index of the array.

The input array **must** be sorted.

Example:

```
use flc_algorithm, only : binary_search, INDEX_INT
implicit none
integer(INDEX_INT) :: idx
integer, dimension(6) :: iarr = [ -5, 1, 1, 2, 4, 9]

idx = binary_search(iarr, -100) ! returns 0
idx = binary_search(iarr, 1)    ! returns 2
idx = binary_search(iarr, 2)    ! returns 4
idx = binary_search(iarr, 3)    ! returns 0
idx = binary_search(iarr, 9)    ! returns 6
idx = binary_search(iarr, 10)   ! returns 0
```

4.1.2.2 equal_range

Finds the range of elements in a sorted array equivalent to the given value. If the exact value isn't present, the first index will point to the index at which the value could be inserted to maintain a sorted array. If searching for a value that's in the sorted array more than once, the expression `arr(first_idx:last_idx)` will return the equal values. If the value isn't present, `arr(first_idx:last_idx)` will be an empty array, and the first index will be the point at which the element would be located if it were present.

Example:

```
use flc_algorithm, only : equal_range, INDEX_INT
implicit none
integer(INDEX_INT) :: first, last
integer, dimension(6) :: iarr = [ -5, 1, 1, 2, 4, 9]

call equal_range(iarr, -6, first, last) ! (first,last) are (1,0)
call equal_range(iarr, -5, first, last) ! (first,last) are (1,1)
call equal_range(iarr,  1, first, last) ! (first,last) are (2,3)
call equal_range(iarr,  3, first, last) ! (first,last) are (5,4)
call equal_range(iarr,  9, first, last) ! (first,last) are (6,6)
```

4.1.2.3 minmax_element

Finds the smallest and largest element in an array. Note that the *first* occurrence of the minimum value is selected, and the *last* occurrence of the maximum value is selected. Thus, for a sorted array `arr` which may have duplicate elements, the expression `arr(min_idx:max_idx)` will always return the entire array.

Example:

```
use flc_algorithm, only : minmax_element, INDEX_INT
implicit none
integer, dimension(6) :: iarr = [ -5, 1000, -1000, 999, -1000, 1000]
integer(INDEX_INT) :: min_idx, max_idx

call minmax_element(iarr, min_idx, max_idx) ! min_idx == 3, max_idx == 6
```

4.1.3 SET OPERATIONS

Sorted arrays can be manipulated as "sets," supporting unions, intersections, and differences.

4.1.3.1 includes

Whether one set encloses another set: every item of the second array is present in the first array.

Example:

```
use flc_algorithm, only : includes
implicit none
integer, dimension(6) :: iarr = [ -5, 1, 2, 4, 9]
integer, dimension(3) :: jarr = [ 1, 2, 5]
logical :: is_superset
```

(continues on next page)

(continued from previous page)

```
is_superset = includes(iarr, iarr)) ! true
is_superset = includes(iarr, iarr(:3))) ! true
is_superset = includes(iarr, iarr(3:))) ! true
is_superset = includes(iarr(3:), iarr)) ! false
is_superset = includes(iarr, jarr) ! false
is_superset = includes(iarr, jarr(1:2))) ! true
```

4.1.3.2 Not yet implemented

- set_difference
- set_intersection
- set_symmetric_difference
- set_union

4.1.4 MODIFYING

4.1.4.1 shuffle

The "shuffle" subroutine depends on the Random module so that it can use the default random number generator to randomly reorder an array.

Example:

```
use flc_algorithm, only : shuffle
use flc_random, only : Engine => MersenneEngine4
implicit none
integer :: i
integer, dimension(8) :: iarr = (/ ((i), i = -4, 3) /)
type(Engine) :: rng
rng = Engine()

call shuffle(rng, iarr)
```

4.1.4.2 Not yet implemented

- unique

4.2 CHRONO

Time calculations and timers are not yet implemented.

4.3 MAP

Maps are sorted dictionaries mapping keys to values. Currently they have limited functionality and few instantiations: maps of ints to ints and of strings to strings.

4.3.1 BASIC FUNCTIONALITY

All map types support the following basic operations.

4.3.1.1 Construction and destruction

Like other wrapped C++ classes in Flibcpp, maps are constructed using an interface function. The default constructor is an empty map. Maps are destroyed using the `release` type-bound subroutine.

4.3.1.2 Modification

The contents of the map can be changed with the following methods:

`insert`: Add a new key-value pair to the map. If the key already exists, the value in the map will remain unchanged. An optional `logical` parameter can be passed that will be set to `.true.` if insertion was successful and `.false.` if the key already existed.

`set`: Assign the given value to the key, regardless of whether the value already existed.

`get`: Return the value for the specified key, creating it with a default value (zero for numeric types, empty for string types) if it does not exist.

`clear`: Remove all items from the map.

The `size` method returns the number of elements, and `count` will return the number of elements with the given key.

Here's an example of creating, modifying, and destroying a map:

```
use flc_map, only : Map => MapIntInt
type(Map) :: m
logical :: inserted = .false.
integer(C_INT) :: value
m = Map()
call m%insert(123, 456)
call m%insert(123, 567, inserted) ! inserted = false, value unchanged
call m%set(123, 567) ! value is changed
value = m%get(911) ! implicitly created value of zero
call m%erase(911)
call m%release()
```

4.3.1.3 Iteration

Iterating over a map to determine its contents is not yet supported.

4.4 RANDOM

The `flc_random` module contains advanced pseudo-random number generation from the random C++ header.

The C++11 way of generating random numbers takes some getting used to. Rather than having a single global function that returns a random real number in the range [0, 1), C++11 has independent *engine* objects that generate streams of random bits. Different *distribution* objects convert those bits into samples of a distribution.

Although C++11 defines a dizzying variety of random number engines, `flc_random` wraps just two: the 32- and 64-bit *Mersenne Twister* algorithms. The 32-bit version (`MersenneEngine4`) is currently the only engine type that can be used with distributions and algorithms.

Flibcpp wraps distribution objects as independent subroutines. Each subroutine accepts the constructor parameters of the distribution, the random engine, and a target Fortran array to be filled with random numbers.

Generating an array with 10 normally-distributed reals with a mean of 8 and a standard deviation of 2 is done as such:

```
use flc_random, only : Engine => MersenneEngine4, normal_distribution
real(C_DOUBLE), dimension(20) :: arr
type(Engine) :: rng

rng = Engine()
call normal_distribution(8.0d0, 2.0d0, rng, arr)
call rng%release()
```

4.4.1 ENGINES

The two Mersenne twister engines in `flc_random` return different-sized integers per call:

- `MersenneEngine4`: each invocation returns a 32-bit `integer(4)`
- `MersenneEngine8`: each invocation returns a 64-bit `integer(8)`

Engines can be constructed using one of two interface functions: the argument-free `MersenneEngine4()` uses the default seed, and the engine takes a single argument `MersenneEngine4(1234567)` with the seed. Alternatively, the seed can be set (or reset) using the `seed()` type-bound procedure.

Generally, engines are used with distributions (described below). However, if necessary, individual randomly generated values can be obtained by calling the `next()` type-bound procedure.

> **Warning:** C++ generates *unsigned* integers with entropy in every bit. This means that the integers obtained from `engine%next()`, reinterpreted as signed Fortran integers, may be negative.

In most cases, the default distribution-compatible `MersenneEngine4` should be used, since the distributions described below require it.

4.4.2 DISTRIBUTIONS

Distributions produce numerical values from the random bitstream provided by an RNG engine. For efficiency, each distribution subroutine accepts an *array* of values that are filled with samples of the distribution.

4.4.2.1 normal_distribution

Each element of the sampled array is distributed according to a Gaussian function with the given mean and standard deviation.

4.4.2.2 uniform_int_distribution

Each element is uniformly sampled between the two provided bounds, inclusive on both sides.

4.4.2.3 uniform_real_distribution

Each element is a sample of a uniform distribution between the two bounds, inclusive on left side only.

4.4.2.4 discrete_distribution

The discrete distribution is constructed with an array of N weights: the probability that an index in the range $[1, N]$ will be selected.

```
real(C_DOUBLE), dimension(4), parameter :: weights &
   = [.125d0, .125d0, .25d0, .5d0]
integer(C_INT), dimension(1024) :: sampled
call discrete_distribution(weights, Engine(), sampled)
```

In the above example, `1` and `2` will are expected to each occupy an eighth of the `sampled` array, approximately a quarter of the `sampled` array's values will be `3`, and about a half will be `4`.

Note: The C++ distribution returns values in $[0, N)$, so in accordance with Flibcpp's indexing convention the result is transformed when provided to Fortran users.

4.5 SET

Sets are sorted containers of unique elements. The `flc_set` module defines sets of `integer` and of `type(String)`.

4.5.1 BASIC FUNCTIONALITY

All set types support the following basic operations.

4.5.1.1 Construction and destruction

Like other wrapped C++ classes in Flibcpp, sets are constructed using an interface function. The default constructor is an empty set. Sets are destroyed using the `release` type-bound subroutine.

4.5.1.2 Modification

The two primary operations on a set are `insert` and `erase` for adding an element to and removing an element from the set. A `clear` subroutine removes all elements from the set.

The `size` method returns the number of elements, and `count` will return the number of elements of a given value.

Here's an example of creating, modifying, and destroying a set:

```
use flc_set, only : Set => SetInt
type(Set) :: s
logical :: inserted
s = Set()
```

(continues on next page)

(continued from previous page)

```
call s%insert(2)
call s%insert(3, inserted) ! Set has 2 elements, inserted => true
call s%insert(3, inserted) ! Duplicate element, ignored; inserted => false
call s%erase(2) ! Remove 2 from the set
call s%erase(1) ! Nonexistent set element, ignored
write(0,*) "Number of 3s in the set:" s%count(3)
call s%clear() ! Remove all elements, size is now zero
call s%insert(1)
call s%release() ! Free memory
```

4.5.1.3 Set operations

The Fortran `Set` classes have been extended to include several useful set algorithms. (In C++, these are implemented using the <algorithm> header and therefore should resemble the functions in the flc_algorithm module.

All set operations take a single argument, another `Set` object, and do not modify either the original or the argument. All but the `includes` return newly allocated `Set` instances and do not modify the original sets.

difference: $A \setminus B$ Returns a new set with all elements from the original that are *not* present in the other set.

intersection: $A \cap B$ Return all elements that are in both sets.

symmetric_difference: $(A \setminus B) \cup (B \setminus A)$ Return all elements that are in one set or the other but not both.

union: $A \cup B$ Return all elements that are in either set.

includes: $A \supseteq B$ Return whether all elements of the other set are in the original set.

4.5.1.4 Iteration

Iterating over a set to determine its contents is not yet supported.

4.5.2 NUMERIC SETS

Unlike vectors, the `flc_set` module includes a single "native integer" numeric instantiations. The value type is `integer(C_INT)` and is 64 bits on most modern systems. Since the C++ implementation of numerical sets is not very efficient, the assumption is that the `set` will be used in a non-numerically-intensive capacity where the default integer is the most appropriate option.

4.5.2.1 Construct from an array

Numeric sets can be created very efficiently from Fortran data by accepting an array argument:

```
use flc_set, only : Set => SetInt
type(Set) :: s

s = Set([1, 1, 2, 10])
write(0,*) "Size should be 3:", s%size()
```

The `assign` bound method acts like a constructor but for an existing set.

4.5.3 STRING SETS

The native "element" type of `SetString` is a `character(len=:)`. Set operations that accept an input will take any native character string; and returned values will be allocatable character arrays.

An additional `insert_ref` function allows assignment of String types

4.6 STRING

The string module includes the `String` derived type and a handful of string conversion functions.

4.6.1 STRING TYPE

The C++ standard library "string" is a dynamically resizable, mutable character array.

4.6.1.1 Constructors

Strings are constructed using three interface functions:

- The function without arguments creates an empty string;
- An integer argument `count` and a single character will create a string of size `count` filled with that character; and
- A standard Fortran `character(kind=C_CHAR, len=*)` which will be copied to the string.

Here are three examples of initialization:

```
use flc_string, only : String
type(String) :: s

s = String()
! s%size() == 0
s = String(10, "!")
! s%size() == 10
! s%get(i) == "!"
s = String("I am a string!")
```

4.6.1.2 Character element access

The number of characters in the string is returned by the bound function `size`. The `get` function returns the character at an index; and `front` and `back` are aliases for `get(1)` and `get(v%size())`, respectively.

Important: Unlike the C++ version of this class, **strings in Flibcpp use 1-offset indexing**. See Indexing.

4.6.1.3 Modification

Like vectors, Strings can be resized dynamically using a variety of methods:

- `resize` to specify an exact size;
- `push_back` to add a new character to the end of it;
- `append` to add another string to the end

- `pop_back` to remove the last character;
- `clear` to remove all elements.

The string also has a `set` bound subroutine for assigning a character to a specified index:

```
type(String) :: s

s = String("=", 10)
call s%set(1, "8")
call s%set(s%size(), "D")
```

4.6.1.4 Search

The `find` bound function will search for a substring, starting at an optional position. Like the search algorithms in Flibcpp, a search result of *0* indicates "not found" and any other result is the 1-offset index in the string.

```
type(String) :: s
integer :: i

s = String("meowmeow")
i = s%find("meow")    ! Returns 1
i = s%find("meow", 3) ! Returns 5
i = s%find("woof")    ! Returns 0
```

4.6.1.5 View as an array pointer

The string can be viewed (and indeed modified) as an array of character elements:

```
type(String) :: s
character, dimension(:), pointer :: charptr

s = String("Hello!")
charptr => s%view()
charptr(6) = "?" ! change greeting to a question
```

4.6.1.6 Conversion to native string

The `str` type-bound function returns an allocated character string:

```
character(len=:), allocatable :: native
type(String) :: s

s = String("Hello world")
native = s%str()
write(0,"(a)") native
```

4.6.2 CONVERSION FUNCTIONS

The `flc_string` module includes several module procedures for converting native Fortran strings to integers and real numbers. These functions are robust and exception-safe, allowing intelligent error handling from the Fortran side.

- Integer conversion: `stoi`, `stol`, `stoll`
- Real conversion: `stof`, `stod`

```
use flc, only : ierr, get_serr, SWIG_OverflowError, SWIG_ValueError
use flc_string
implicit none
integer(4) :: temp
character(len=100) :: tempstr

read(*, '(a)') tempstr
temp = stoi(trim(tempstr))
if (ierr == SWIG_OverflowError) then
  write(0,*) "Your integer is too darn big!"
elseif (ierr == SWIG_ValueError) then
  write(0,*) "That thing you entered? It wasn't an integer."
end if
```

Integer conversion defaults to base-10, but passing an additional integer argument allows conversion from other bases. The special integer value of `0` allows auto-detection of values in octal (with a leading `0` as in `0777`) or hexadecimal (with a leading `0x` as in `0xb1f1c2a3`).

4.7 VECTOR

Vectors are resizeable arrays of elements. The `flc_vector` module instantiates vectors of `integer(4)`, `integer(8)`, `real(8)`, `complex(8)`, and `type(String)`.

4.7.1 COMMON FUNCTIONALITY

All vector types support the following basic operations.

4.7.1.1 Construction and destruction

Vectors are constructed using four interface functions:

- The function without arguments creates an empty vector;
- A single integer argument assigns that many elements with default values; and
- An integer argument followed by an element with the vector's element type will copy that value to all elements of the vector.
- A vector object will create a copy of that vector.

Here are three examples of initialization:

```
use flc_vector, only : Vector => VectorInt4
type(Vector) :: v

v = Vector()
! v%size() == 0
v = Vector(10)
! v%size() == 10
! v%get(i) == 0
v = Vector(10, 123)
! v%size() == 10
! v%get(i) == 123
```

Vectors are destroyed using the `release` type-bound subroutine:

```
call v%release()
```

4.7.1.2 Modification

Vectors can be resized dynamically using `resize`, which acts like the constructors described above. An element can be added to the end of the vector (increasing the size by one) with `push_back`. The `insert` method can insert an element at a specific index, and `erase` removes a specific vector index or range of indices. `clear` removes all elements. Finally, `set` sets the value of an element at a given index.

Important: Unlike the C++ version of this class, **all vectors in Flibcpp use 1-offset indexing**. This means that `v%get(1)` is the same as the C++ `v[0]`: it returns the first element (i.e. the element with an offset of zero).

Here's an example of modifying a vector:

```
use flc_vector, only : Vector => VectorInt4
type(Vector) :: v
v = Vector()
call v%resize(4, 123) ! give each element the value 123
call v%push_back(-1) ! size increased by 1, last element has value -1
call v%insert(2, -2) ! First 3 elements are [123, 123, -2]
call v%erase(1, 3) ! Remove the first two elements
call v%erase(2) ! Remove the second element
call v%set(1, -123) ! Change the value of the first element
call v%clear() ! Remove all elements, size is now zero
```

4.7.1.3 Access

The size of a vector is returned by the bound function `size`; `get` returns the value at an index; and `front` and `back` are aliases for `get(1)` and `get(v%size())`, respectively.

Additionally, `front_ref`, `back_ref`, and `get_ref` return Fortran pointers to the elements of the array.

> **Warning:** Array element pointers are valid **only** as long as the vector's size is not changed. Calling `erase`, `push_back`, and so forth will invalidate the pointer; accessing it at that point results in undefined behavior.

4.7.2 NUMERIC VECTORS

As with the algorithms and other methods, the `flc_vector` module includes three scalar numeric instantiations, but it also includes an instantiation for complex numbers. Each instantiation has a distinct derived type:

- `VectorInt4`: each element is `integer(4)`
- `VectorInt8`: each element is `integer(8)`
- `VectorReal8`: each element is `real(8)`
- `VectorComplex8`: each element is `complex(8)`

4.7.2.1 Construct from an array

Numeric vectors can be created very efficiently from Fortran data by accepting an array pointer:

```
use flc_vector, only : Vector => VectorInt4
integer(4), dimension(4), parameter :: iarr = [ 1, -2, 4, -8 ]
type(Vector) :: v

v = Vector(iarr)
write(0,*) "Size should be 4:", v%size()
```

The `assign` bound method acts like a constructor but for an existing vector.

4.7.2.2 View as an array pointer

Numeric vectors can also return an array pointer to the vector's contents. These views support native Fortran array operations and access the same underlying memory as the C++ object:

```
use flc_vector, only : Vector => VectorInt4
integer(4), dimension(:), pointer :: vptr
type(Vector) :: v

! <snip>
vptr => v%view()
if (size(vptr) > 2) then
   vptr(2) = 4
end if
```

Warning: A vector's view is valid **only** as long as the vector's size is not changed. Calling `erase`, `push_back`, and so forth will invalidate the view; accessing it at that point results in undefined behavior.

4.7.3 STRING VECTORS

The native "element" type of `VectorString` is a `character(len=:)`. Vector operations that accept an input will take any native character string; and returned values will be allocatable character arrays.

The `front_ref`, `back_ref`, and `get_ref` functions allow the underlying `std::string` class to be accessed with the `String` Fortran derived type wrapper. Note that unlike for intrinsic types, where these functions return a `integer, pointer`, the vector of strings returns just `type(String)`. However, as with native pointers described above, these references are *invalid* once the string changes size. They should be cleared with the `%release()` bound method.

An additional `set_ref` function allows vector elements to be assigned from `String` types.

5. EXAMPLES

The following standalone codes demonstrate how Flibcpp can be used in native Fortran code.

5.1 RANDOM NUMBERS AND SORTING

This simple example generates an array of normally-distributed double-precision reals, sorts them, and then shuffles them again.

```
!-----------------------------------------------------------------------------!
! \file   example/sort.f90
!
! Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
!-----------------------------------------------------------------------------!

program sort_example
  use, intrinsic :: ISO_C_BINDING
  use flc
  use flc_algorithm, only : sort, shuffle
  use flc_random, only : Engine => MersenneEngine4, normal_distribution
  use example_utils, only : write_version, read_positive_int, STDOUT
  implicit none
  integer :: arr_size
  real(c_double), dimension(:), allocatable :: x
  real(c_double), parameter :: MEAN = 1.0d0, SIGMA = 0.5d0
  type(Engine) :: rng

  ! Print version information
  call write_version()

  ! Get array size
  arr_size = read_positive_int("array size")
  allocate(x(arr_size))

  ! Fill randomly with normal distribution
  rng = Engine()
  call normal_distribution(MEAN, SIGMA, rng, x)

  ! Sort the array
  call sort(x)
  write(STDOUT, "(a, 4(f8.3,','))") "First few elements:", x(:min(4, size(x)))

  ! Rearrange it randomly
  call shuffle(rng, x)
  write(STDOUT, "(a, 4(f8.3,','))") "After shuffling:", x(:min(4, size(x)))

```

(continues on next page)

(continued from previous page)

```
  call rng%release()
end program

!------------------------------------------------------------------------------!
! end of example/sort.f90
!------------------------------------------------------------------------------!
```

5.2 VECTORS OF STRINGS

Strings and vectors of strings can be easily manipulated and converted to and from native Fortran strings.

```
!------------------------------------------------------------------------------!
! \file   example/vecstr.f90
!
! Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
!------------------------------------------------------------------------------!

program vecstr_example
  use, intrinsic :: ISO_C_BINDING
  use flc
  use flc_string, only : String
  use flc_vector, only : VectorString
  use example_utils, only : read_strings, write_version, STDOUT
  implicit none
  integer :: i
  type(VectorString) :: vec
  type(String) :: back, front, temp
  character(C_CHAR), dimension(:), pointer :: chars

  ! Print version information
  call write_version()

  ! Read a vector of strings
  call read_strings(vec)

  write(STDOUT, "(a, i3, a)") "Read ", vec%size(), " strings:"
  do i = 1, vec%size()
    write(STDOUT, "(i3, ': ', a)") i, vec%get(i)
  end do

  if (vec%empty()) then
    write(STDOUT, *) "No vectors provided"
    call vec%release()
    stop 0
  endif

```

(continues on next page)

(continued from previous page)

```
  ! Get the final string for modification
  back = vec%back_ref()
  chars => back%view()
  temp = String(back%str())
  ! Change all characters to exclamation points
  chars(:) = '!'
  write(STDOUT, *) "The last string is very excited: " // vec%get(vec%size())

  ! Modify a reference to the front value
  front = vec%front_ref()
  call front%push_back("?")

  ! Insert the original 'back' after the first string (make it element #2)
  call vec%insert(2, temp%str())
  ! Inserting the vector invalidates the 'chars' view and back reference.
  chars => NULL()
  back = vec%back_ref()
  write(STDOUT, *) "Inserted the original last string: " // vec%get(2)

  ! Modify back to be something else.
  call back%assign("the end")

  write(STDOUT, *) "Modified 'front' string is " // vec%get(1)
  write(STDOUT, *) "Modified 'back' string is " // vec%get(vec%size())

  ! Remove the first string (invalidating back and front references)
  call vec%erase(1)
  call back%release()
  call front%release()

  write(STDOUT, "(a, i3, a)") "Ended up with ", vec%size(), " strings:"
  do i = 1, vec%size()
    write(STDOUT, "(i3, ': ', a)") i, vec%get(i)
  end do

  ! Free allocated vector memory
  call vec%release()
end program

!-----------------------------------------------------------------------------!
! end of example/sort.f90
!-----------------------------------------------------------------------------!
```

5.3 GENERIC SORTING

Since sorting algorithms often allow $O(N)$ algorithms to be written in $O(\log N)$, providing generic sorting routines is immensely useful in applications that operate on large chunks of data. This example demonstrates the generic version of the argsort subroutine by sorting a native Fortran array of native Fortran types using a native Fortran subroutine. The only C interaction needed is to create C pointers to the Fortran array entries and to provide a C-bound comparator that converts those pointers back to native Fortran pointers.[1]

```
!------------------------------------------------------------------------------!
! \file   example/sort_generic.f90
!
! Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
!------------------------------------------------------------------------------!

! Mock-up of a user-created type and comparison operator
module sort_generic_extras
  implicit none
  public

  ! Declare an example Fortran derived type
  type :: FortranString
    character(len=:), allocatable :: chars
  end type

  ! Declare a 'less than' operator for that type
  interface operator(<)
    module procedure fortranstring_less
  end interface

contains

! Lexicographically compare strings of equal length.
elemental function chars_less(left, right, length) &
    result(fresult)
  character(len=*), intent(in) :: left
  character(len=*), intent(in) :: right
  integer, intent(in) :: length
  logical :: fresult
  integer :: i, lchar, rchar
```

(continues on next page)

[1] Older versions of Gfortran (before GCC-8) fail to compile the generic sort example because of a bug that incorrectly claims that taking the C pointer of a scalar Fortran value is a violation of the standard:

```
../example/sort_generic.f90:84:38:

     call c_f_pointer(cptr=rcptr, fptr=rptr)
                                      1
Error: TS 29113/TS 18508: Noninteroperable array FPTR at (1) to
C_F_POINTER: Expression is a noninteroperable derived type
```

See this bug report for more details.

(continued from previous page)

```
  ! If any character code is less than the RHS, it is less than.
  do i = 1, length
    lchar = ichar(left(i:i))
    rchar = ichar(right(i:i))
    if (lchar < rchar) then
      fresult = .true.
      return
    elseif (lchar > rchar) then
      fresult = .false.
      return
    endif
  end do

  fresult = .false.
end function

elemental function fortranstring_less(self, other) &
    result(fresult)
  type(FortranString), intent(in) :: self
  type(FortranString), intent(in) :: other
  logical :: fresult

  if (.not. allocated(other%chars)) then
    ! RHS is null and LHS is not
    fresult = .true.
  elseif (.not. allocated(self%chars)) then
    ! LHS is null => "greater than" (if LHS is string) or equal (if both null)
    fresult = .false.
  elseif (len(self%chars) < len(other%chars)) then
    ! Since LHS is shorter, it is "less than" the RHS.
    fresult = .true.
  elseif (len(self%chars) > len(other%chars)) then
    ! If RHS is shorter
    fresult = .false.
  else
    ! Compare strings of equal length
    fresult = chars_less(self%chars, other%chars, len(self%chars))
  endif
end function

! C++-accessible comparison function for two pointers-to-strings
! (null strings always compare "greater than" to move to end of a list)
function compare_strings(lcptr, rcptr) bind(C) &
    result(fresult)
  use, intrinsic :: ISO_C_BINDING
```

(continues on next page)

(continued from previous page)

```
  type(C_PTR), intent(in), value :: lcptr
  type(C_PTR), intent(in), value :: rcptr
  logical(C_BOOL) :: fresult
  type(FortranString), pointer :: lptr
  type(FortranString), pointer :: rptr

  if (.not. c_associated(rcptr)) then
    ! RHS is null and LHS is not
    fresult = .true.
  elseif (.not. c_associated(lcptr)) then
    ! LHS is null => "greater than" (if LHS is string) or equal (if both null)
    fresult = .false.
  else
    ! Both associated: convert from C to Fortran pointers
    call c_f_pointer(cptr=lcptr, fptr=lptr)
    call c_f_pointer(cptr=rcptr, fptr=rptr)

    ! Compare the strings
    fresult = (lptr < rptr)
  endif
end function
end module

program sort_generic_example
  use, intrinsic :: ISO_FORTRAN_ENV
  use, intrinsic :: ISO_C_BINDING
  use flc
  use flc_algorithm, only : argsort, INDEX_INT
  use sort_generic_extras, only : compare_strings, FortranString
  use example_utils, only : write_version, read_positive_int, STDOUT, STDIN
  implicit none
  type(FortranString), dimension(:), allocatable, target :: fs_array
  type(C_PTR), dimension(:), allocatable, target :: ptrs
  integer(INDEX_INT), dimension(:), allocatable, target :: ordering
  character(len=80) :: readstr
  integer :: arr_size, i, io_ierr

  call write_version()

  ! Read strings
  arr_size = read_positive_int("string array size")
  allocate(fs_array(arr_size))
  do i = 1, arr_size
    write(STDOUT, "(a, i3)") "Enter string #", i
    read(STDIN, "(a)", iostat=io_ierr) readstr
    if (io_ierr == IOSTAT_END) then
```

(continues on next page)

(continued from previous page)

```
      ! Leave further strings unallocated
      exit
    endif
    ! Allocate string
    allocate(fs_array(i)%chars, source=trim(readstr))
  enddo

  ! Create C pointers to the Fortran objects
  ptrs = [(c_loc(fs_array(i)), i = 1, arr_size)]

  ! Use 'argsort' to determine the new ordering
  allocate(ordering(arr_size))
  call argsort(ptrs, ordering, compare_strings)
  write(STDOUT, "(a, 20(i3))") "New order:", ordering

  ! Reorder the Fortran data
  fs_array = fs_array(ordering)

  ! Print the results
  write(STDOUT, *) "Sorted:"
  do i = 1, arr_size
    if (.not. allocated(fs_array(i)%chars)) then
      write(STDOUT, "(i3, '-', i3, a)") i, arr_size, " are unallocated"
      exit
    endif
    write(STDOUT, "(i3, ': ', a)") i, fs_array(i)%chars
  enddo

end program

!-----------------------------------------------------------------------------!
! end of example/sort.f90
!-----------------------------------------------------------------------------!
```

5.4 EXAMPLE UTILITIES MODULE

This pure-Fortran module builds on top of functionality from Flibcpp. It provides procedures to:

- Format and print the Flibcpp version;
- Converts a user input to an integer, validating it with useful error messages;
- Reads a dynamically sized vector of strings from the user.

```
!-----------------------------------------------------------------------------!
! \file   example/example_utils.f90
! \brief  example_utils module
! \note   Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
```

(continues on next page)

(continued from previous page)

```
!------------------------------------------------------------------------------!

module example_utils
  use, intrinsic :: ISO_FORTRAN_ENV
  use, intrinsic :: ISO_C_BINDING
  implicit none
  integer, parameter :: STDOUT = OUTPUT_UNIT, STDIN = INPUT_UNIT
  public

contains

subroutine write_version()
  use flc
  implicit none
  ! Print version information
  write(STDOUT, "(a)") "========================================"
  write(STDOUT, "(a, a)") "Flibcpp version: ", get_flibcpp_version()
  write(STDOUT, "(a, 2(i1,'.'), (i1), a)") "(Numeric version: ", &
      flibcpp_version_major, flibcpp_version_minor, flibcpp_version_patch, &
      ")"
  write(STDOUT, "(a)") "========================================"
end subroutine

! Loop until the user inputs a positive integer. Catch error conditions.
function read_positive_int(desc) result(result_int)
  use flc
  use flc_string, only : stoi
  implicit none
  character(len=*), intent(in) :: desc
  character(len=80) :: readstr
  integer :: result_int, io_ierr
  do
    write(STDOUT, *) "Enter " // desc // ": "
    read(STDIN, "(a)", iostat=io_ierr) readstr
    if (io_ierr == IOSTAT_END) then
      ! Error condition: ctrl-D during input
      write(STDOUT, *) "User terminated"
      stop 1
    endif

    result_int = stoi(readstr)
    if (ierr == 0) then
      if (result_int <= 0) then
        ! Error condition: non-positive value
        write(STDOUT, *) "Invalid " // desc // ": ", result_int
        continue
```

(continues on next page)

(continued from previous page)

```
      end if

      write(STDOUT, *) "Read " // desc // "=", result_int
      exit
    endif

    if (ierr == SWIG_OVERFLOWERROR) then
      ! Error condition: integer doesn't fit in native integer
      write(STDOUT,*) "Your integer is too darn big!"
    else if (ierr == SWIG_VALUEERROR) then
      ! Error condition: not an integer at all
      write(STDOUT,*) "That text you entered? It wasn't an integer."
    else
      write(STDOUT,*) "Unknown error", ierr
    end if
    write(STDOUT,*) "(Detailed error message: ", get_serr(), ")"

    ! Clear error flag so the next call to stoi succeeds
    ierr = 0
  end do
end function

! Loop until the user inputs a positive integer. Catch error conditions.
subroutine read_strings(vec)
  use flc
  use flc_string, only : String
  use flc_vector, only : VectorString
  use ISO_FORTRAN_ENV
  implicit none
  type(VectorString), intent(out) :: vec
  integer, parameter :: STDOUT = OUTPUT_UNIT, STDIN = INPUT_UNIT
  character(len=80) :: readstr
  integer :: io_ierr
  type(String) :: str

  ! Allocate the vector
  vec = VectorString()

  do
    ! Request and read a string
    write(STDOUT, "(a, i3, a)") "Enter string #", vec%size() + 1, &
        " or Ctrl-D/empty string to complete"
    read(STDIN, "(a)", iostat=io_ierr) readstr
    if (io_ierr == IOSTAT_END) then
      ! Break out of loop on ^D (EOF)
      exit
```

(continues on next page)

(continued from previous page)

```
    end if

    ! Add string to the end of the vector
    call vec%push_back(trim(readstr))
    ! Get a String object reference to the back to check if it's empty
    str = vec%back_ref()
    if (str%empty()) then
      ! Remove the empty string
      call vec%pop_back()
      exit
    end if
  end do
end subroutine

end module

!-------------------------------------------------------------------------------!
! end of example/example_utils.f90
!-------------------------------------------------------------------------------!
```

REFERENCES

[Bea03] D.M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, July 2003. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167739X02001711 (visited on 2018-09-06), doi:10.1016/S0167-739X(02)00171-1.

[JPE20] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated Fortran-C++ Bindings for Large-Scale Scientific Applications. *Computing in Science & Engineering*, 22(5):84–94, October 2020. URL: https://ieeexplore.ieee.org/document/8745480/ (visited on 2019-08-20), doi:10.1109/MCSE.2019.2924204.

[MHD+21] Lois Curfman McInnes, Michael A. Heroux, Erik W. Draeger, Andrew Siegel, Susan Coghlan, and Katie Antypas. How community software ecosystems can unlock the potential of exascale computing. *Nature Computational Science*, 1(2):92–94, February 2021. URL: http://www.nature.com/articles/s43588-021-00033-y (visited on 2021-05-22), doi:10.1038/s43588-021-00033-y.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Appendix A. INTERFACE

Appendix A. INTERFACE

These are the SWIG interface files used to generate the Flibcpp modules.

A.1 FLC

The primary file defines typemaps.

```
/*!
 * \file flc.i
 *
 * Copyright (c) 2019-2020 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc"

#if defined(SWIGIMPORTED) && !defined(FLC_SWIGIMPORTED)
#error "To import the FLC module correctly, use ``%include \"import_flc.i\"``"
#endif

/* ------------------------------------------------------------------------
 * Header definition macros
 * ------------------------------------------------------------------------ */

%define %flc_add_header
%insert("fbegin") %{
! Flibcpp project, https://github.com/swig-fortran/flibcpp
! Copyright (c) 2019-2020 Oak Ridge National Laboratory, UT-Battelle, LLC.
! Distributed under an MIT open source license: see LICENSE for details.
%}
%insert("begin") %{
/*
 * Flibcpp project, https://github.com/swig-fortran/flibcpp
 * Copyright (c) 2019-2020 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */
%}
%enddef

%flc_add_header

/* ------------------------------------------------------------------------
 * Exception handling
 * ------------------------------------------------------------------------ */

// Rename the error variables' internal C symbols
#define SWIG_FORTRAN_ERROR_INT flc_ierr
```

(continues on next page)

(continued from previous page)

```
#define SWIG_FORTRAN_ERROR_STR flc_get_serr

// Restore names in the wrapper code
%rename(ierr) flc_ierr;
%rename(get_serr) flc_get_serr;

// Unless we're directly building this module, delay exception handling
#ifndef SWIGIMPORTED
%include <exception.i>
#endif

/* ------------------------------------------------------------------------
 * Data types and instantiation
 * ------------------------------------------------------------------------ */

// Note: stdint.i inserts #include <stdint.h>
%include <stdint.i>

%define %flc_template_numeric(SRC, DST)
%template(DST) SRC<int32_t>;
%template(DST) SRC<int64_t>;
%template(DST) SRC<double>;
%enddef

/* ------------------------------------------------------------------------
 * Array view translation
 * ------------------------------------------------------------------------ */

%include <typemaps.i>
%apply (SWIGTYPE *DATA, size_t SIZE) {
       (int32_t  *DATA, size_t DATASIZE),
       (int64_t  *DATA, size_t DATASIZE),
       (double   *DATA, size_t DATASIZE),
       (void    **DATA, size_t DATASIZE)};

%apply (SWIGTYPE const *DATA, size_t SIZE) {
       (int32_t  const *DATA, size_t DATASIZE),
       (int64_t  const *DATA, size_t DATASIZE),
       (double   const *DATA, size_t DATASIZE),
       (void*    const *DATA, size_t DATASIZE)};

/* ------------------------------------------------------------------------
 * Version information
 * ------------------------------------------------------------------------ */

%apply char* { const char flibcpp_version[] };
```

(continues on next page)

(continued from previous page)

```
%fortranbindc flibcpp_version_major;
%fortranbindc flibcpp_version_minor;
%fortranbindc flibcpp_version_patch;

// These symbols are defined in the CMake-generated `flibcpp_version.cpp`
%inline %{
extern "C" {
extern const char flibcpp_version[];
extern const int flibcpp_version_major;
extern const int flibcpp_version_minor;
extern const int flibcpp_version_patch;
}
%}
```

A.2 FLC_ALGORITHM

```
/*!
 * \file flc_algorithm.i
 *
 * Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_algorithm"
%include "import_flc.i"
%flc_add_header

%{
#include <algorithm>
#include <functional>
#include <numeric>
%}

/* ------------------------------------------------------------------------
 * Macros
 * ------------------------------------------------------------------------ */
%define %flc_cmp_algorithm(RETURN_TYPE, FUNCNAME, ARGS, CALL)

%inline {
// Operate using default "less than"
template<class T>
static RETURN_TYPE FUNCNAME(ARGS) {
  return FUNCNAME##_impl(CALL, std::less<T>());
}
// Operate using user-provided function pointer
template<class T>
```

(continues on next page)

(continued from previous page)

```
static RETURN_TYPE FUNCNAME##_cmp(ARGS, bool (*cmp)(T, T)) {
  return FUNCNAME##_impl(CALL, cmp);
}
}

// Instantiate numeric overloads
%flc_template_numeric(FUNCNAME, FUNCNAME)
%flc_template_numeric(FUNCNAME##_cmp, FUNCNAME)

// Instantiate comparators with void* arguments
%template(FUNCNAME) FUNCNAME##_cmp<void*>;

%enddef

/* ------------------------------------------------------------------------ */
%define %flc_typemaps(NAME, TYPE...)

// Apply array conversion typemap
%apply (const SWIGTYPE *DATA, size_t SIZE) {
    (TYPE const *DATA1, size_t DATASIZE1),
    (TYPE const *DATA2, size_t DATASIZE2) };

// Explicitly declare function interface for callbacks
%fortrancallback("%s") flc_cmp_##NAME;
extern "C" bool flc_cmp_##NAME(TYPE left, TYPE right);

%enddef

/* ------------------------------------------------------------------------
 * Types
 * ------------------------------------------------------------------------ */

// Alias the native C integer to an "indexing" integer returned by algorithm
// functions.
%inline %{
typedef int index_int;
%}
%insert("fdecl") %{integer, parameter, public :: INDEX_INT = C_INT
%}

// Give it a particularly named type in the Fortran proxy code.
%apply int { index_int };
%typemap(ftype, in={integer(INDEX_INT), intent(in)}) index_int
  %{integer(INDEX_INT)%}

// Apply array-to-C translation for numeric values
```

(continues on next page)

(continued from previous page)

```
%apply (SWIGTYPE *DATA, size_t SIZE) { (index_int *IDX, size_t IDXSIZE) };

// Apply array and callback typemaps
%flc_typemaps(int4 , int32_t   )
%flc_typemaps(int8 , int64_t   )
%flc_typemaps(real8, double    )
%flc_typemaps(index, index_int )
%flc_typemaps(ptr  , void*     )

/* ------------------------------------------------------------------------
 * Sorting routines
 * ------------------------------------------------------------------------ */

%{
template<class T, class Compare>
static void sort_impl(T *data, size_t size, Compare cmp) {
    return std::sort(data, data + size, cmp);
}

template<class T, class Compare>
static bool is_sorted_impl(const T *data, size_t size, Compare cmp) {
    return std::is_sorted(data, data + size, cmp);
}

template<class T, class Compare>
static void argsort_impl(const T *data, size_t size,
                         index_int *index, size_t index_size,
                         Compare cmp) {
  // Fill invalid indices with zero
  if (size < index_size) {
    std::fill(index + size, index + index_size, 0);
  }
  size = std::min(size, index_size);
  // Fill the indices with 1 through size
  std::iota(index, index + size, 1);
  // Define a comparator that accesses the original data
  auto int_sort_cmp = [cmp, data](index_int left, index_int right)
  { return cmp(data[left - 1], data[right - 1]); };
  // Let the standard library do all the hard work!
  std::sort(index, index + size, int_sort_cmp);
}

%}

%flc_cmp_algorithm(void, sort,
                   %arg(T *DATA, size_t DATASIZE),
```

(continues on next page)

(continued from previous page)

```
                    %arg(DATA, DATASIZE))
%flc_cmp_algorithm(bool, is_sorted,
                    %arg(const T *DATA, size_t DATASIZE),
                    %arg(DATA, DATASIZE))
%flc_cmp_algorithm(void, argsort,
                    %arg(const T *DATA, size_t DATASIZE,
                         index_int *IDX, size_t IDXSIZE),
                    %arg(DATA, DATASIZE, IDX, IDXSIZE))

/* ------------------------------------------------------------------------
 * Searching routines
 * ------------------------------------------------------------------------ */

%{
template<class T, class Compare>
static index_int binary_search_impl(const T *data, size_t size, T value,
                                    Compare cmp) {
  const T *end = data + size;
  auto iter = std::lower_bound(data, end, value, cmp);
  if (iter == end || cmp(*iter, value) || cmp(value, *iter))
    return 0;
  // Index of the found item *IN FORTAN INDEXING*
  return (iter - data) + 1;
}

template<class T, class Compare>
static void equal_range_impl(const T *data, size_t size, T value,
                             index_int &first_index, index_int &last_index,
                             Compare cmp) {
  const T *end = data + size;
  auto range_pair = std::equal_range(data, end, value, cmp);
  // Index of the min/max items *IN FORTAN INDEXING*
  first_index = range_pair.first - data + 1;
  last_index = range_pair.second - data;
}

template<class T, class Compare>
static void minmax_element_impl(const T *data, size_t size,
                                index_int &min_index, index_int &max_index,
                                Compare cmp) {
  const T *end = data + size;
  auto mm_pair = std::minmax_element(data, end, cmp);
  // Index of the min/max items *IN FORTAN INDEXING*
  min_index = mm_pair.first - data + 1;
  max_index = mm_pair.second - data + 1;
}
```

(continues on next page)

(continued from previous page)

```
%}

%flc_cmp_algorithm(index_int, binary_search,
                   %arg(const T *DATA, size_t DATASIZE, T value),
                   %arg(DATA, DATASIZE, value))

%flc_cmp_algorithm(void, equal_range,
                   %arg(const T *DATA, size_t DATASIZE, T value,
                        index_int &first_index, index_int &last_index),
                   %arg(DATA, DATASIZE, value, first_index, last_index))

%flc_cmp_algorithm(void, minmax_element,
                   %arg(const T *DATA, size_t DATASIZE,
                        index_int &min_index, index_int &max_index),
                   %arg(DATA, DATASIZE, min_index, max_index))

/* ------------------------------------------------------------------------
 * Set operation routines
 * ------------------------------------------------------------------------ */

%{
template<class T, class Compare>
static bool includes_impl(const T *data1, size_t size1,
                          const T *data2, size_t size2,
                          Compare cmp) {
  return std::includes(data1, data1 + size1, data2, data2 + size2, cmp);
}
%}

%flc_cmp_algorithm(bool, includes,
                   %arg(const T *DATA1, size_t DATASIZE1,
                        const T *DATA2, size_t DATASIZE2),
                   %arg(DATA1, DATASIZE1, DATA2, DATASIZE2))

/* ------------------------------------------------------------------------
 * Modifying routines
 * ------------------------------------------------------------------------ */

%{
#include <random>
%}

%import "flc_random.i"

%inline {
template<class T>
```

(continues on next page)

(continued from previous page)

```
static void shuffle(std::FLC_DEFAULT_ENGINE& g, T *DATA, size_t DATASIZE) {
    std::shuffle(DATA, DATA + DATASIZE, g);
}
}

%flc_template_numeric(shuffle, shuffle)
%template(shuffle) shuffle<void*>;
```

A.3 FLC_CHRONO

```
/*!
 * \file flc_chrono.i
 *
 * Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_chrono"
%include "import_flc.i"
%flc_add_header

/* ------------------------------------------------------------------------
 * Utility routines
 * ------------------------------------------------------------------------ */

%{
#include <chrono>
#include <thread>
#include <stdexcept>
%}

%inline %{
static void sleep_for(int ms) {
    if (ms < 0)
        throw std::domain_error("Invalid sleep time");
    std::this_thread::sleep_for(std::chrono::milliseconds(ms));
}
%}
```

A.4 FLC_RANDOM

```
/*!
 * \file flc_random.i
 *
 * Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_random"
%include "import_flc.i"
%flc_add_header

%{
#include <random>
#if defined(_MSC_VER) && _MSC_VER < 1900
// Visual studio 2012's standard library lacks iterator constructors for
// std::discrete_distribution
#define FLC_MISSING_DISCRETE_ITER
#endif
%}

/* ------------------------------------------------------------------------
 * Macros
 * ------------------------------------------------------------------------ */

%define %flc_random_engine(NAME, GENERATOR, RESULT_TYPE)
namespace std {

%rename(NAME) GENERATOR;
%rename("next") GENERATOR::operator();

class GENERATOR
{
  public:
    typedef RESULT_TYPE result_type;

    GENERATOR();
    explicit GENERATOR(result_type seed_value);
    void seed(result_type seed_value);
    void discard(unsigned long long count);
    result_type operator()();
};

} // namespace std
%enddef

```

(continues on next page)

(continued from previous page)

```
/* ------------------------------------------------------------------------
 * RNG distribution routines
 * ------------------------------------------------------------------------ */

%{
template<class D, class G, class T>
static inline void flc_generate(D dist, G& g, T* data, size_t size) {
  T* const end = data + size;
  while (data != end) {
    *data++ = dist(g);
  }
}
%}

%apply (const SWIGTYPE *DATA, size_t SIZE) {
       (const double *WEIGHTS, size_t WEIGHTSIZE) };

%inline %{
template<class T, class G>
static void uniform_int_distribution(T left, T right,
                                     G& engine, T* DATA, size_t DATASIZE) {
  flc_generate(std::uniform_int_distribution<T>(left, right),
               engine, DATA, DATASIZE);
}

template<class T, class G>
static void uniform_real_distribution(T left, T right,
                                      G& engine, T* DATA, size_t DATASIZE) {
  flc_generate(std::uniform_real_distribution<T>(left, right),
               engine, DATA, DATASIZE);
}

template<class T, class G>
static void normal_distribution(T mean, T stddev,
                                G& engine, T* DATA, size_t DATASIZE) {
  flc_generate(std::normal_distribution<T>(mean, stddev),
               engine, DATA, DATASIZE);
}

template<class T, class G>
static void discrete_distribution(const double* WEIGHTS, size_t WEIGHTSIZE,
                                  G& engine, T* DATA, size_t DATASIZE) {
#ifndef FLC_MISSING_DISCRETE_ITER
  std::discrete_distribution<T> dist(WEIGHTS, WEIGHTS + WEIGHTSIZE);
#else
  std::discrete_distribution<T> dist(
```

(continues on next page)

(continued from previous page)

```
      std::initializer_list<double>(WEIGHTS, WEIGHTS + WEIGHTSIZE));
#endif
  T* const end = DATA + DATASIZE;
  while (DATA != end) {
    *DATA++ = dist(engine) + 1; // Note: transform to Fortran 1-offset
  }
}
%}

%define %flc_distribution(NAME, STDENGINE, TYPE)
%template(NAME##_distribution) NAME##_distribution< TYPE, std::STDENGINE >;
%enddef

// Engines
%flc_random_engine(MersenneEngine4, mt19937,    int32_t)
%flc_random_engine(MersenneEngine8, mt19937_64, int64_t)

#define FLC_DEFAULT_ENGINE mt19937
%flc_distribution(uniform_int,  FLC_DEFAULT_ENGINE, int32_t)
%flc_distribution(uniform_int,  FLC_DEFAULT_ENGINE, int64_t)
%flc_distribution(uniform_real, FLC_DEFAULT_ENGINE, double)

%flc_distribution(normal, FLC_DEFAULT_ENGINE, double)

// Discrete sampling distribution
%flc_distribution(discrete, FLC_DEFAULT_ENGINE, int32_t)
%flc_distribution(discrete, FLC_DEFAULT_ENGINE, int64_t)
```

A.5 FLC_SET

```
/*!
 * \file flc_set.i
 *
 * Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_set"
%include "import_flc.i"
%flc_add_header

%include <std_set.i>

// Support for set operations
%{
#include <algorithm>
```

(continues on next page)

(continued from previous page)

```
#include <iterator>
%}

/* -----------------------------------------------------------------------
 * Macro definitions
 * ----------------------------------------------------------------------- */

%define %flc_define_set_algorithm(FUNCNAME)
  %insert("header") {
  template<class Set_t>
  static Set_t flc_##FUNCNAME(const Set_t& left, const Set_t& right)
  {
      Set_t result;
      std::FUNCNAME(left.begin(), left.end(),
                    right.begin(), right.end(),
                    std::inserter(result, result.end()));
      return result;
  }
  } // end %insert
%enddef

%define %flc_extend_set_algorithm(FUNCNAME, RETVAL, TYPE)
  // The rename with the stringifying macro is necessary because 'union' is a
  // keyword.
  %rename(#FUNCNAME) std::set<TYPE>::set_##FUNCNAME;
  %extend std::set<TYPE> {
   RETVAL set_##FUNCNAME(const std::set<TYPE>& other)
   { return flc_set_##FUNCNAME(*$self, other); }
  } // end %extend
%enddef

%define %flc_std_set_extend_pod(CTYPE)
%extend {
  %apply (const SWIGTYPE *DATA, ::size_t SIZE)
    { (const CTYPE* DATA, size_type SIZE) };

  // Construct from an array of data
  set(const CTYPE* DATA, size_type SIZE) {
    return new std::set<CTYPE>(DATA, DATA + SIZE);
  }

  // Insert an array of data
  void insert(const CTYPE* DATA, size_type SIZE) {
    $self->insert(DATA, DATA + SIZE);
  }
}
```

(continues on next page)

(continued from previous page)

```
%enddef

/* ------------------------------------------------------------------------ */
/*! \def %specialize_std_set_pod
 *
 * Inject member functions and typemaps for POD classes.
 *
 * These provide an efficient constructor from a Fortan array view.
 *
 * This definition is considered part of the \em public API so that downstream
 * apps that generate FLC-based bindings can instantiate their own POD sets.
 */
%define %specialize_std_set_pod(T)

namespace std {
  template<> class set<T> {
    %swig_std_set(T, std::less<T>, std::allocator<T>)
    %flc_std_set_extend_pod(T)
  };
}
%enddef

/* ------------------------------------------------------------------------
 * Algorithms
 * ------------------------------------------------------------------------ */

%flc_define_set_algorithm(set_difference)
%flc_define_set_algorithm(set_intersection)
%flc_define_set_algorithm(set_symmetric_difference)
%flc_define_set_algorithm(set_union)

%insert("header") %{
template<class Set_t>
static bool flc_set_includes(const Set_t& left, const Set_t& right)
{
    return std::includes(left.begin(), left.end(),
                         right.begin(), right.end());
}
%}

%define %flc_extend_algorithms(TYPE)
  %flc_extend_set_algorithm(difference, std::set<TYPE >, TYPE)
  %flc_extend_set_algorithm(intersection, std::set<TYPE >, TYPE)
  %flc_extend_set_algorithm(symmetric_difference, std::set<TYPE >, TYPE)
  %flc_extend_set_algorithm(union, std::set<TYPE >, TYPE)
  %flc_extend_set_algorithm(includes, bool, TYPE)
```

(continues on next page)

(continued from previous page)

```
%enddef

/* ------------------------------------------------------------------------
 * Numeric sets
 * ------------------------------------------------------------------------ */

%flc_extend_algorithms(int)
%specialize_std_set_pod(int)

%template(SetInt) std::set<int>;

/* ------------------------------------------------------------------------
 * String sets
 * ------------------------------------------------------------------------ */

// Allow direct insertion of a wrapped std::string
%extend std::set<std::string> {
  %apply SWIGTYPE& { const std::string& STR_CLASS };

  void insert_ref(const std::string& STR_CLASS) {
    $self->insert(STR_CLASS);
  }
}

%include <std_string.i>
%import "flc_string.i"
%flc_extend_algorithms(std::string)
%template(SetString) std::set<std::string>;
```

A.6 FLC_STRING

```
/*!
 * \file flc_string.i
 *
 * Copyright (c) 2019 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_string"
%include "import_flc.i"
%flc_add_header

// SWIG always represents std::string as native strings. We load its typemaps
// but will explicitly create the class.
%include <std_string.i>

```

(continues on next page)

(continued from previous page)

```
// Include typemaps for integer offsets and native integer types
%include <std_common.i>

/* -----------------------------------------------------------------------
 * Typemaps
 * ----------------------------------------------------------------------- */

// Typemap to convert positions from npos -> 0 and 1-offset otherwise. Similar
// to
%apply int FORTRAN_INT { std::size_t POSITION };
%typemap(out, noblock=1) std::size_t POSITION {
  $result = ($1 == std::string::npos ? 0 : $1 + 1);
}

/* -----------------------------------------------------------------------
 * String class definition
 * ----------------------------------------------------------------------- */

namespace std {
class string {
  public:
    // >>> TYPES
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef char value_type;
    typedef const char& const_reference;

    // Typemaps for making std::vector feel more like native Fortran:
    // - Use Fortran 1-offset indexing
    %apply int FORTRAN_INDEX {size_type pos,
                              size_type index,
                              size_type start_index,
                              size_type stop_index};
    // - Use native Fortran integers in proxy code
    %apply int FORTRAN_INT {size_type};

    // - Use fortran indexing (and 0 for not found) for search
    %apply std::size_t POSITION {size_type find};

    // - Allow access as an array view
    %apply SWIGTYPE& { string& view };
    %fortran_array_pointer(char, string& view);
    %typemap(out, noblock=1) string& view {
      $result.data = ($1->empty() ? NULL : const_cast<char*>($1->data()));
      $result.size = $1->size();
    }
```

(continues on next page)

(continued from previous page)

```
62
63      // - Allow interaction with other string objects
64      %apply SWIGTYPE& {const string& OTHER};
65
66    public:
67      // >>> MEMBER FUNCTIONS
68
69      string();
70      string(size_type count, value_type ch);
71      string(const std::string& s);
72
73      // Accessors
74      size_type size() const;
75      bool empty() const;
76
77      const_reference front() const;
78      const_reference back() const;
79
80      // Modify
81      void resize(size_type count);
82      void resize(size_type count, value_type v);
83      void assign(const string& s);
84      void push_back(value_type v);
85      void pop_back();
86      void clear();
87
88      // String operations
89      size_type find(const string& s, size_type pos = 0);
90      void append(const string& s);
91      int compare(const string& OTHER);
92
93      // >>> EXTENSIONS
94
95      %extend {
96        %fragment("SWIG_check_range");
97
98        void set(size_type index, value_type v) {
99          SWIG_check_range(index, $self->size(),
100                           "std::string::set",
101                           return);
102         (*$self)[index] = v;
103       }
104
105       value_type get(size_type index) {
106         SWIG_check_range(index, $self->size(),
107                          "std::string::get",
```

(continues on next page)

(continued from previous page)

```
                        return $self->front());
        return (*$self)[index];
      }

      // Get a character array view
      string& view() { return *$self; }

      // Get a copy as a native Fortran string
      const string& str() { return *$self; }
    }
};

/* -----------------------------------------------------------------------
 * String conversion routines
 * ----------------------------------------------------------------------- */

%exception {
  SWIG_check_unhandled_exception();
  try {
    $action
  }
  catch (const std::invalid_argument& e) {
    SWIG_exception(SWIG_ValueError, e.what());
  }
  catch (const std::out_of_range& e) {
    SWIG_exception(SWIG_OverflowError, e.what());
  }
}

%fragment("flc_has_junk", "header",
          fragment="<cctype>", fragment="<algorithm>") %{
  SWIGINTERN bool flc_has_junk(const std::string& s, size_t pos) {
    return !std::all_of(s.begin() + pos, s.end(),
                        [](unsigned char c) -> bool { return std::isspace(c); });
  }
%}

%typemap(in, numinputs=0, noblock=1) size_t* result_pos (size_t temp_pos) {
  temp_pos = 0;
  $1 = &temp_pos;
}
%typemap(argout, noblock=1, fragment="flc_has_junk") size_t* result_pos {
  if (flc_has_junk(*arg1, temp_pos$argnum)) {
    SWIG_exception(SWIG_ValueError, "Junk at end of string");
  }
}
```

(continues on next page)

(continued from previous page)

```

// String conversion routines
#define %add_string_int_conversion(RETURN_TYPE, NAME) \
  RETURN_TYPE NAME(const string& s, size_t* result_pos, int base = 10)
#define %add_string_real_conversion(RETURN_TYPE, NAME) \
  RETURN_TYPE NAME(const string& s, size_t* result_pos)

%add_string_int_conversion(int, stoi);
%add_string_int_conversion(long, stol);
%add_string_int_conversion(long long, stoll);
%add_string_real_conversion(float, stof);
%add_string_real_conversion(double, stod);

// Don't add exception code for subsequent functions
%exception;

} // namespace std
```

A.7 FLC_VECTOR

```
/*!
 * \file flc_vector.i
 *
 * Copyright (c) 2019-2020 Oak Ridge National Laboratory, UT-Battelle, LLC.
 * Distributed under an MIT open source license: see LICENSE for details.
 */

%module "flc_vector"
%include "import_flc.i"
%flc_add_header

%include <complex.i>
%include <std_vector.i>

/* ---------------------------------------------------------------------------
 * Macro definitions
 * ------------------------------------------------------------------------- */

%define %flc_std_vector_extend_pod(CTYPE, IMTYPE)
%extend {
  %apply (const SWIGTYPE *DATA, ::size_t SIZE)
    { (const CTYPE* DATA, size_type SIZE) };

  // Construct from an array of data
  vector(const CTYPE* DATA, size_type SIZE) {
    return new std::vector<CTYPE>(DATA, DATA + SIZE);
```

(continues on next page)

(continued from previous page)

```
  }

  // Assign from another vector
  void assign(const CTYPE* DATA, size_type SIZE) {
    $self->assign(DATA, DATA + SIZE);
  }

  // Get a mutable view to ourself
  %fortran_array_pointer(IMTYPE, vector<CTYPE>& view);

  %typemap(out, noblock=1) vector<CTYPE>& view {
    $result.data = ($1->empty() ? NULL : &(*$1->begin()));
    $result.size = $1->size();
  }

  vector<CTYPE>& view() {
    return *$self;
  }
}
%enddef

/* ------------------------------------------------------------------------ */
/*! \def %flc_template_std_vector_pod
 *
 * Inject member functions and typemaps for POD classes, and instantiate.
 *
 * The added methods provide an efficient constructor from a Fortan array view.
 * It also offers a "view" functionality for getting an array pointer to the
 * vector-owned data.
 *
 * This definition is considered part of the \em public API so that downstream
 * apps that generate FLC-based bindings can instantiate their own POD vectors.
 */
%define %flc_template_std_vector_pod(NAME, T)

namespace std {
  template<> class vector<T> {

    %swig_std_vector(T, const T&)
    %swig_std_vector_extend_ref(T)
    %flc_std_vector_extend_pod(T, T)
  };
}

// Instantiate the template
%template(NAME) std::vector<T>;
```

(continues on next page)

(continued from previous page)

```
73
74  %enddef
75
76
77  /* ------------------------------------------------------------------------ */
78  /*! \def %flc_template_std_vector_complex
79   *
80   * Inject member functions and typemaps for std::complex instantiations.
81   *
82   * This definition is considered part of the \em public API so that downstream
83   * apps that generate FLC-based bindings can instantiate their own POD vectors.
84   */
85  %define %flc_template_std_vector_complex(NAME, T)
86
87  namespace std {
88    template<> class vector<std::complex<T> > {
89
90      %swig_std_vector(std::complex<T>, const std::complex<T>&)
91      %swig_std_vector_extend_ref(std::complex<T>)
92      %flc_std_vector_extend_pod(std::complex<T>, SwigComplex_##T)
93    };
94  }
95
96  // Instantiate the template
97  %template(NAME) std::vector<std::complex<T> >;
98
99  %enddef
100
101 /* ------------------------------------------------------------------------
102  * Numeric vectors
103  * ------------------------------------------------------------------------ */
104
105 %flc_template_std_vector_pod(VectorInt4,  int32_t)
106 %flc_template_std_vector_pod(VectorInt8,  int64_t)
107 %flc_template_std_vector_pod(VectorReal8, double)
108
109 %flc_template_std_vector_complex(VectorComplex8, double)
110
111 /* ------------------------------------------------------------------------
112  * String vectors
113  * ------------------------------------------------------------------------ */
114
115 %include <std_string.i>
116 %import "flc_string.i"
117
118 %apply SWIGTYPE& { const std::string& value };
```

(continues on next page)

(continued from previous page)

```
119
120 %extend std::vector<std::string> {
121   void set_ref(size_type index, const std::string& value) {
122     SWIG_check_range(index, $self->size(),
123                      "std::vector<std::string>::set_ref",
124                      return);
125     (*$self)[index] = value;
126   }
127 }
128
129 %template(VectorString) std::vector<std::string>;
130
131 %clear const std::string& value;
```

Appendix B. LICENSE

Appendix B. LICENSE