Chombo Support for Particles

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

May 1, 2002

Contents

1 Overview of API Design

We make use of templating and inheritance to integrate particles into the existing Chombo infrastructure.

2 Classes for Respresenting Particles (ParticleTools)

The class `BinItem` is the basic particle base class from which all other particles can be defined, and contains the interface through which other classes can interact with it. A `BinFab` is derived from BaseFab, and contains a list of `BinItems` in each grid cell. An auxiliary container class `List<T>` has been added to BoxTools to facilitate this [1]

[1] Two points here –

- Neither the `Vector` or the `List` classes are documented in the main Chombo Design document

2.1 Class `BinItem`

The `BinItem` class is the base class for a single particle. It is intended that specific particle implementations will be contained in classes derived from `BinItem`. As the most basic particle type, it contains a position, along with functions for accessing and setting the position. Any other particle characteristics (charge, velocity, etc) will be part of a specific particle implementation derived from `BinItem`. The important member functions of `BinItem` are as follows:

- `BinItem();`

  Default constructor

- `virtual void define(const RealVect& a_position);`

  Initialize particle to position `a_position`.

- `void setPosition(const RealVect& a_position)`

  set particle position to `a_position`

- `void setPosition(const Real a_position, const int a_dimension)`

  set position component `a_dimension` to `a_position`

- `RealVect& position();`

  Returns reference to this `BinItem`'s position, which may be modified.

- `const RealVect& position() const;`

  Returns const reference to this `BinItem`'s position.

- `Real position(const int a_dir) const;`

  Returns this `BinItem`'s position component in `a_dir`.

- `virtual int size() const;`

  Returns the size, in number of bytes, of a flat linear representation of the data in this object.

- `virtual void linearOut(void* buf) const;`

  Write a linear representation of the internal data of this `BinItem` class; assumes that sufficient memory for the buffer has already been allocated by the caller.

- `virtual void linearIn(void* buf);`

  Read linear representation of the data needed to define this `BinItem` class from the buffer. Any existing data is overwritten.

- Attribution may be an issue here – `List` is actually a resurrected BoxLib class

2.2 Class BinFab

BinFab is a class for holding and sorting particle-type items, derived from BaseFab. A BinFab<T> is an enhanced BaseFab<List<T> >, where the class <T> must have a RealVect <T>::position() const function which is used to place items in the appropriate bins. The important member functions of BinFab are:

- BinFab()

  Default constructor.

- ```
  BinFab(const Box& a_domain,
         const RealVect& a_mesh_spacing,
         const RealVect& a_origin,
         const ProblemDomain& a_probdomain);
  ```

  and

  ```
  void define(const Box& a_domain,
              const RealVect& a_mesh_spacing,
              const RealVect& a_origin,
              const ProblemDomain& a_probdomain);
  ```

  Defines this BinFab over the box given by a_domain. a_mesh_spacing and a_origin define the size and location of the bins into which particles will be sorted. The BinFab defined by this function will be empty (it will contain no particles).

- BinFab(const BinFab& a_src);

  Copy constructor – copies contents of a_src to newly created BinFab

- virtual void reBin();

  Sorts particles in this BinFab into the correct bins. Particles which no longer reside within any of the bins in this BinFab are eliminated. (It is assumed that if they are moving to another BinFab, the user will have done this before the reBin function is called.

- virtual void addItems(const List<T>& a_list);

  Items in a_list are sorted into bins in this BinFab. Items which are not located within bins in this BinFabs domain are ignored.

- virtual void addItemsDestructive(List<T>& a_list);

  Similar to the addItems function, except as items are placed in this BinFab, they are removed from a_list. When this function returns, a_list only contains those
  ```

items which were not placed in this BinFab. [2].

- `virtual void clear()`

  Return this BinFab to an undefined state.

- `virtual int size(const Box& a_box, const Interval& a_comps) const;`

  This function returns the size, in number of bytes, of a flat representation of the data in this object contained by the sub-box a_box over the components a_comps.

- ```
  virtual void linearOut(void* buf,
                         const Box& R,
                         const Interval& comps) const
  ```

  Write a linear representation of the internal data of this BinFab class over the sub-domain R and components comps; assumes that sufficient memory for the buffer has already been allocated by the caller.

- ```
  virtual void linearIn(void* buf,
                        const Box& R,
                        const Interval& comps)
  ```

  Read linear representation of the data needed to define this BinItem class from the buffer. Any existing data is overwritten.

2.3 Class BinFabFactory

The BinFabFactory class is a factory class to produce BinFab's, and is derived from the DataFactory base class. The important functions in this class are:

- ```
  BinFabFactory(const RealVect& a_mesh_spacing,
                const RealVect& a_origin,
                const ProblemDomain& a_domain);
  ```

  Creates a BinFabFactory and fills internal data with inputs which will be used to define any BinFab created using the create function.

- ```
  virtual BinFab<T>* create(const Box& a_box, int a_ncomps,
                            const DataIndex& a_dit) const;
  ```

  Creates a new BinFab object and returns a pointer to it. Responsiblitly for calling operator 'delete' on this pointer is passed to the user.

[2] This function not yet implemented, but it will be once I get a chance. If somebody has a better name for this function, let me know

2.4 The `List` and `ListIterator` Classes

The `List<T>` class is a doubly-linked list container class for objects of type `T`, while the `ListIterator<T>` class allows access to the contents of a `List<T>`.

The documentation for these classes should eventually go into the Chombo documentation. Another issue is that these classes could possibly stand to be revised to make them fit in better with the Chombo way of doing things. They are presented here because they are needed for the ParticleTools classes.

2.4.1 List Functions

Important functions for the `List` class are:

- `List()`

  Construct an empty `List<T>`

- `List(const List<T> a_rhs)`

  Copy constructor

- `List<T>& operator= (const List<T>& a_rhs);`

  The assignment operator

- `void prepend(const T& a_value);`

  Adds a copy of the value to the beginning of the List

- `void append (const T& value);`

  Adds a copy of the value to the end of the `List<T>`.

- `void add (const T& value);`

  Adds a copy of the value to the end of the `List<T>`.

- `void join (const List<T>& src);`

  Appends a copy of all items in `List<T>` src to this `List<T>`.

- `void catenate (List<T>& src);`

  Appends a copy of all items in `List<T>` src to this `List<T>`. This differs from join() in that it unlinks the objects from the `List<T>` src and glues them to the end of this `List<T>`, leaving `List<T>` src empty. This is more efficient that join() if src is no longer needed.

- `void clear()`

  Removes all objects from the `List<T>`.

- `inline List<T>* copy () const;`

  Returns a copy of this `List<T>` on the heap. It is the user's responsibility to delete this when no longer needed.

- `inline T& firstElement () const;`

  Returns a reference to the first element in the `List<T>`.

- `inline T& lastElement () const;`

  Returns a reference to the last element in the `List<T>`.

- `bool includes (const T& value) const;`

  Returns true if the `List<T>` contains an object identical to value. Type T must have an operator==() defined, or be an intrinsic type.

- `bool operator== (const List<T>& rhs) const;`

  Returns true if the this and rhs are memberwise equal; i.e. the two lists are the same size and each of the elements in the list compare equal. Type T must have an operator==() defined, or be an intrinsic type.

- `bool operator!= (const List<T>& rhs) const;`

  Returns true if the this and rhs are not equal.

- `inline bool isEmpty () const;`

  Returns true if the `List<T>` is empty.

- `inline bool isNotEmpty () const;`

  Returns true if the `List<T>` is not empty.

- `int length () const;`

  Returns the number of objects in the `List<T>`.

- `inline void removeFirst ();`

  Removes the first element in the `List<T>`.

- `inline void removeLast ();`

  Removes the last element in the `List<T>`.

- `inline const T& operator[] (const ListIterator<T>& li) const;`

  Returns reference to object pointed to by the `ListIterator<T>`.

- `inline T& operator[] (const ListIterator<T>& li);`

  Returns reference to object pointed to by the `ListIterator<T>`.

- `void remove (const T& value);`

  Removes all objects in the `List<T>` equal to value.

- `void remove (const List<T>& lst);`

  Removes all objects in the `List<T>` equal to any of the values in lst.

- `void remove (ListIterator<T>& lit);`

  Removes the object pointed to by the `ListIterator<T>`.

- ```
  inline void replace (ListIterator<T>& li,
                       const T&         val);
  ```

  Replace the value pointed to by the `ListIterator<T>` by val.

- ```
  inline void addAfter (ListIterator<T>& lit,
                        const T&         val);
  ```

  Insert val into `List<T>` after the object pointed to by `ListIterator<T>`.

- ```
  inline void addBefore (ListIterator<T>& lit,
                         const T&         val);
  ```

  Insert val into `List<T>` before the object pointed to by `ListIterator<T>`.

- `inline ListIterator<T> first () const;`

  Returns a `ListIterator<T>` to the first object in this `List<T>`.

- `inline ListIterator<T> last () const;`

  Returns a `ListIterator<T>` to the last object in this `List<T>`.

2.4.2 ListIterator Functions

Important functions for the `ListIterator` class are:

- `inline ListIterator (const List<T>& aList);`

  Construct a `ListIterator<T>` to first element of aList.

- `inline ListIterator (const ListIterator<T>& rhs);`

  The copy constructor.

- `inline void rewind ();`

  Reset this `ListIterator<T>` to point to the first element in the `List<T>` .

- `inline const T& operator() () const;`

  Return a constant reference to the object in the `List<T>` currently pointed to by this `ListIterator<T>` .
  ```

- `inline const T& operator* () const;`

  Return a constant reference to the object in the `List<T>` currently pointed to by this `ListIterator<T>` .

- `inline operator bool () const;`

  This is a conversion operator that makes the iterator look like a pointer. This operator makes it easy to check if the iterator is pointing to an element on the `List<T>` . If the iterator has been moved off the `List<T>` or if the `List<T>` is empty, this conversion returns the NULL pointer.

- `inline bool operator! () const;`

  Returns true if `ListIterator<T>` doesn't point to any element on the `List<T>` .

- `inline const T& value () const;`

  Return a constant reference to the object in the `List<T>` currently pointed to by the iterator.

- `inline ListIterator<T>& operator++ ();`

  This is the prefix auto-increment operator. It advances the `ListIterator<T>` to point to the next element on the `List<T>` . It then returns a reference to itself to allow for chaining with other operators.

- `inline ListIterator<T>& operator-- ();`

  This is the prefix auto-decrement operator. It moves the `ListIterator<T>` to point to the previous element on the `List<T>` . It then returns a reference to itself to allow for chaining with other operators.

- `inline ListIterator<T> operator-- (int);`

  This is the postfix auto-decrement operator. It moves the `ListIterator<T>` to point to the previous element on the `List<T>` . It then returns a `ListIterator<T>` that points to the old element to allow for chaining with other operators.

- `inline ListIterator<T> operator++ (int);`

  This is the postfix auto-increment operator. It advances the `ListIterator<T>` to point to the next element on the `List<T>` . It then returns a `ListIterator<T>` that points to the old element to allow for chaining with other operators.

- `inline bool operator== (const ListIterator<T>&) const;`

  Do the two `ListIterator<T>` s point to the same `List<T>` and the same element within the `List<T>` ?

- `inline bool operator!= (const ListIterator<T>&) const;`
  Are the `ListIterator<T>` s not equal?

References