SEP Vector: C++ Features in Code

Eileen Martin
Library development done with Bob Clapp, Huy Le, Chris Leader

March 11, 2014

Abstract

Many students in SEP primarily code in Fortran 90, so this document gives an overview of some of the features of the C++ SEP inversion library that are not commonly used in Fortran 90 with the goal of helping more people understand, use, and contribute to the library.

1 Navigating the Source Code Directories

Currently, the top level of the code looks like this

- `base`, a folder containing the source code and unit tests
  - Lots of source code files that should be navigated using the documentation in the `docs` folder
  - `utests`, a folder containing all the unit tests of the source code. This contains some useful examples of how to use the library.
- `config.defs, Makefile` Files used to make the source code into a library in `lib` and `obj`, folders that are automatically created when `make all` is run
- `Doxyfile`, a documentation configuration file automatically generated when the command `doxygen -g Doxyfile` is run in the top level directory.
- `docs`, a folder holding the documentation files generated by Doxygen, which are automatically organized into two subdirectories, `latex` and `html`
  - `handWrittenDoc`, a folder containing this document. Don't delete this folder.
- `gee_examples`, a folder containing examples from [**?**] which are currently being built
  - `inverse_nmo.cpp`, NMO example using NMO operator defined in `Nmo.h` and `Nmo.cpp` ****
  - `regSolver.cpp`, code for the regularized solver from section 3.4 using multi-spaces and multi-operators that combine linear interpolation from `lint1.cpp/h` and `deriv2nd1D.cpp/h`

1.1 Automatic Documentation

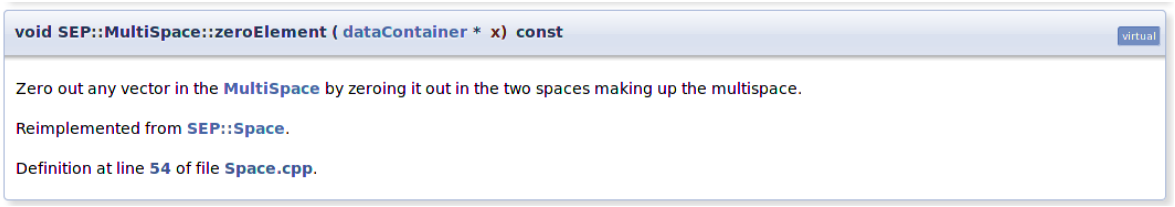
Currently, the documentation is sitting in the `docs` folder. To update the documentation when you add new code or comment code, just run `doxygen Doxyfile`, where Doxyfile is the configuration file. Once this is generated, you can navigate the html documentation by opening `docs/html/index.html` in a browser, or run `docs/latex/refman.tex` through pdflatex to create a reference manual in pdf format.

The documentation that is automatically generated shows namespaces, class hierarchy, inputs and outputs of functions and methods, variables belonging to classes, file structure, and any comments denoted in the code by special markers. One way to create comments that will be included in this documentation is by putting at least two consecutive lines starting with `///` in the implementation of a function or method. For example, this code can be found in `Space.cpp`:

```
void SEP::MultiSpace::zeroElement(dataContainer * x)const{
 /// Zero out any vector in the MultiSpace by zeroing
 /// it out in the two spaces making up the multispace.
 SEP::MultiDataContainer *x2=(SEP::MultiDataContainer*) x;
 sp1->zeroElement(x2->getData1());
```

```
  sp2->zeroElement(x2->getData2());
}
```

The corresponding part of the auto-generated HTML code looks like:



2 Namespaces

Where is it in the code? When browsing the Doxygen documentation, one of the first noticeable features of the code's structure is that literally everything is contained in the SEP namespace. Browsing the source code, nearly every class definition is contained in a `using namespace SEP{}` statement.

Why is it used? If this code is used in isolation, this feature is largely useless [1], but this feature makes it much easier to combine this code with other C++ code. Many of our classes have relatively common names like Vector, Space, Operator, Map. Say that a user wishes to combine the inversion library with a visualization library. There is a good possibility that the visualization library could have a class by the same name, so the user would specify the namespace they reference.

How do I use it? Use the scope resolution operator, ::, as

```
NamespaceName::ClassFunctionOrMethodName
```

to call a function, method or class in the namespace called NamespaceName. When implementing methods of classes in the SEP namespace, follow this example from Operator.cpp, which specifies the domain associated with a SEP::Map:

```
//outputType NamespaceName::ClassName::MethodName(inputs){
void SEP::Map::setDomain(Space *dom){
  domain=dom;
}
```

Going back to the visualization library example, say the user wants to use the inversion library alongside a visualization library which has a class called Vector under the namespace VisLib. The user might wish to plot a SEP::Vector using VisLib and would distinguish between the two types of vectors as follows:

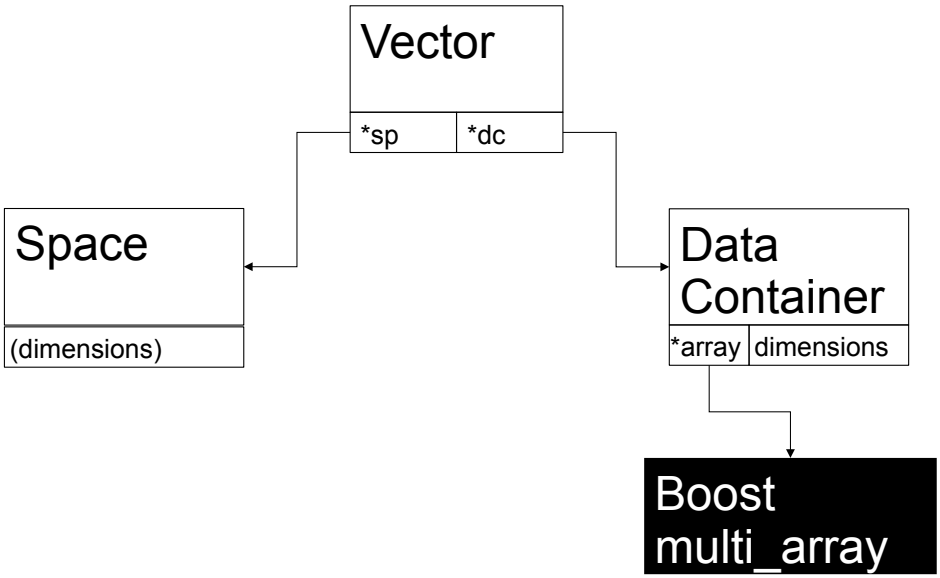
```
SEP::inCoreFloat1D sp(10);     // create sp, 10-D space for in core computing
SEP::Vector vs(&sp);           // create vs, a SEP Vector in the space sp
vs.random();                   // fill vs with ten random numbers
// ... use some SEP namespace methods to manipulate this vector ...
VisLib::Vector vs2(vs);        // create a VisLib Vector, vs2, from vs
vs2.Make1DPlot();              // call a Make1DPlot method of VisLib Vector
```

3 SEP Vectors

The design of SEP::Vectors is inspired by the Rice Vector Library [**?**]. Typically when we hand write code, our operators might have some minimal amount of error checking- something along the lines of asking whether the dimension of the input and output are correct. What happens when two spaces, just happen to have the same dimensions? The user must be very careful to get the

[1] We chose to capitalize Vector and Map with the intention of not overlapping names with the standard C++ namespace which has std::vector and std::map containers.

operator order correct, which can be especially tricky when there is a chain of many operators. When each vector has an associated space, and each operator has a domain space and range space, the software can help users avoid these types of errors.



Each Vector has a pointer to a concretely implemented Space, and a pointer to a concretely implemented dataContainer. Every type of Space implemented thus far has dimension, but it is okay if down the line someone decides to implement a Space that does not have dimensions. Each dataContainer contains some dimensions, and a pointer to a multi-dimensional array which is managed by the Boost multiArray library. Note that the dataContainer is actually created by the Space so these will be compatible. The Boost library has a nice interface, so we can treat it as a black box.

4 Abstract Classes, (Pure) Virtual Methods

What are they? Any base (i.e. parent) class in C++ can have some concretely implemented methods, but any methods that are preceded by the word `virtual` can be redefined in the derived classes. An abstract class is a class that is never meant to be instantiated, but rather serves as a base class to its concretely implemented derived classes. It must have at least one function, known as a *pure virtual* function, which has literally no definition in the base class. These are denoted by a `virtual` marker at the beginning, and a `=0` at the end. These pure virtual functions must be defined in every concrete derived class of the abstract base class. We can then use pointers to abstract base classes to reduce the amount of code, and they are able to be type compatible with their derived classes.

How are they used in the code? Some examples of abstract classes include the dataContainer, Vector and Space classes. Often you will see implementations that simply refer to some `*dataContainer` or `*Space`. When the code is executed, the specific concrete implementation for that type of dataContainer or Space is then carried out. To illustrate, here's an example of the abstract dataContainer class from `dataContainer.h`:

```
class dataContainer{
  public:
    dataContainer(){;}                    // constructor
    virtual void my_type()=0;   // pure virtual fct. makes class abstract
    virtual ~dataContainer(){;}           // destructor
    char name[1000];
  private:                                // no private members
};
```

Here's dataInCoreFloat, an abstract class derived from dataContainer from `DataInCoreFloat.h`:

```
class dataInCoreFloat: public dataContainer {
  public:
    dataInCoreFloat() {};
    void setup_omp();
    std::vector<long long> *get_b_omp(){return &b_omp;} // for vectorization
    std::vector<long long> *get_n_omp(){return &n_omp;}
    float *getFloatPtr(){return vals;}          // for accessing data
    void set_n123(long long n){ n123=n;}        // to set dimensions
    int get_nth(){return nth;}                          // to get dimensions
    void random(dataContainer *x) const;
    virtual void my_type() = 0;          // still keeps class abstract
    virtual ~dataInCoreFloat(){    // need to be able to overwrite
        destructor
      cleanInCoreFloat();
    }
  public:
    float *vals;
  private:
     std::vector<long long> b_omp,n_omp;
     void cleanInCoreFloat();
     long long n123;
     int nth;
     int vector_beg;
}
```

An example of one of the concrete classes derived from dataInCoreFloat is data2DFloat , also defined in DataInCoreFloat.h:

```
class data2DFloat: public dataInCoreFloat{
  public:
    data2DFloat(long long n1, long long n2){
      base_2d(n1,n2);   // create empty n1xn2 boost::multi_array
    }
        data2DFloat(std::vector<long long>n){
      base_2d(n[0],n[1]);
    }
    data2DFloat(long long *n){
      base_2d(n[0],n[1]);
    }
    virtual void *returnData(){return (void*)array;}
    virtual void my_type(){ std::cerr<<"I am dataInCoreFloat2D "<<std::endl
        ;}
    virtual ~data2DFloat(){ delete array;}
  private:
    void base_2d(long long n1,long long n2);
    float_2d *array;
};
```

5 Memory Management

Note that we use raw pointers, so the arrays that are created must be manually freed. In managing the data structures, each concrete dataContainer destructor (denoted by a ClassName() in the class definition) calls some clean method, which deletes the data that the `*array` points to. For example, in DataInCoreFloat.cpp, you see the cleanup method for a data1DFloat :

```
void SEP::data1DFloat::clean_1d(){
        delete array;
        array = NULL;
};
```

Note that any pointers created in the user's code must also be manually cleaned up.

6 The Boost::MultiArray Library

Why use this? The Boost MultiArray library gives an easy interface to access and modify multi-dimensional arrays [?]. It avoids the extra costs associated with creating a matrix as a vector of vectors, or the difficult indexing of a multidimensional array stored as a big single dimensional array in a contiguous block of memory. By using this library, we are able to avoid dealing with some of the details of working with multidimensional array. This library has been tested much more than we would be able to test our own structures. Further, many Linux and Unix distributions already have pre-build Boost packages, and it is easy to download and install, so it will not hinder the portability of this code.

Where is it used? Only the dataContainers ever actually deal with the Boost library. Here's an example from DataInCoreFloat.h, which starts out with renaming the boost::multi_array's.

```
typedef boost::multi_array<float, 1> float_1d;
typedef boost::multi_array<float, 2> float_2d;
// ..... more typedef statements up to 7d.......
// ..... definition of dataInCoreFloat class ....
// ..... definition of data1DFloat through data7DFloat class ...........
```

The example in the previous section shows how data2DFloat is implemented as a concrete class. Later in DataInCoreFloat.cpp, we see the implementation of the `base_2d` call, which calls a Boost constructor:

```
void SEP::data2DFloat::base_2d(long long n1,long long n2){
        array= new float_2d(boost::extents[n1][n2]);
        long long n=n1;
        n*=(long long) n2;
        set_n123(n);                        // sets dimensions of array
        vals=(float*)array->data();    // fills array with data
        setup_omp();                        // sets up blocking of array
}
```

7 Pointer Aliasing and the `__restrict__` keyword

What is pointer aliasing? What happens when two pointer variables with different names point to the same memory location? In Fortran, all pointers are assumed to not be aliased, but In C++, we need to use the `__restrict__` keyword to suggest to the computer that a pointer does not alias any other pointer. This means the compiler is able to make optimizations to the code because it has more information about the independence of two pointers.

Where is it seen? Here is an example of a `__restrict__` in its natural habitat. Remember that when an operation like scaling a Vector in a Space needs to happen, the user's call on the Vectors `vy` and `vx`, `vy = vx.scale(a)`, looks like it just acts on the Vector. Really the operation is implemented as a method of the Space. The `__restrict__` statements below tell the computer that `x` and `y`, the pointers to the dataContainers of `vx` and `vy`, point to data stored in different locations, so it knows it's solving $v_y = a \cdot v_x$ and not $v_x = a \cdot v_x$.

```
void  SEP::inCoreFloat::scale(float a, dataContainer *x,dataContainer *y)
    const {
   dataInCoreFloat *s=(dataInCoreFloat*)x;
   dataInCoreFloat *t=(dataInCoreFloat*)y;
   std::vector<long long> *b_omp=s->get_b_omp(),*n_omp=s->get_n_omp();
   float *__restrict__ my=t->getFloatPtr();
   float *__restrict__ their=s->getFloatPtr();
   // First few entries of vector
   for(long long i=0; i < b_omp->at(0); i++)  my[i]=their[i]*a;
   int ith;
   #pragma omp parallel
   ith=omp_get_thread_num();
   long long j=0;
   SIMD_constant c0;
   SIMD_Float16::setConstant(c0,a);
   SIMD_Float16 mine;
   // Vectorized calls in blocks
   for(long long i=b_omp->at(ith); j < n_omp->at(ith); j++, i+=BLOCK_SIZE){
```

```
      mine.loadu(&their[i]);
      mine*=c0;
      mine.stream(&my[i]);
    }
    // Finish it up
    int nth=s->get_nth()-1;
    for(long long i=b_omp->at(nth)+n_omp->at(nth); i< n123; i++) my[i]=their[
        i]*a;
}
```

8 Error Handling with Exceptions

Why is it used? This is a way of dealing with errors that occur in a certain chunk of code, enclosed in a `try { }` statement. If the code encounters some bad conditions, it may throw an exception. Following this part of the code is a `catch { }` statement, which handles that exception, perhaps printing out information about what type of exception was thrown. After an exception is caught, the computer continues to execute the code after the `catch { }` block. More simplistic error handling might just stop execution after encountering a problem, but the use of exceptions allows the code to continue if possible.

Where is it seen? Exception handling is seen in many of the unit tests, but also in Space.h where the equality of two spaces is checked:

```
virtual bool operator ==(const Space & otherSpace) const
{
        try{
                if (this == &otherSpace) return true;
                throw "MultiSpace don't match";
        }
        catch (const char *xx) {
                throw SEPException(xx);
        }
}
```

References

[1] Ronald Garcia, Jeremy Siek, Andrew Lumsdaine, http://www.boost.org/doc/libs/1_55_0/libs/multi_array/doc/index.html

[2] Jon Claerbout, Sergey Fomel, Last update March 3, 2014, http://sepwww.stanford.edu/sep/prof/gee/book-sep.pdf

[3] William Symes et. al., The Rice Vector Library, http://trip.rice.edu/software/rvl/doc/html/index.html