B2000++ – A C++ implementation of B2000

Thomas Ludwig
SMR Engineering
tludwig@smr.ch

November 24, 2000

B2000++ is a study of the feasibility and usefulness of a C++-based implementation of B2000

Contents

1 Motivation 2

2 The Benefits and Disadvantages of Exceptions 3

3 Overview of B2000++ 5

4 Introduction to the Kernel Classes 5

5 Adding Element Types 7

6 The Helper Classes 8
6.1 ID's 8
6.2 ElementTypeID 8
6.3 ElementParam 9
6.4 Field 9
6.5 Container Classes 9

7 A detailed look at the ElementTypeSet Class 9

8 Storage 12

9 Implementation 13

10 Performance 13

11 Conclusion 13

1 Motivation

The use of a programming language is tied to many factors: Ease of use, experience, availability, and of course, performance. When B2000 was created, the programming language that could best fulfill the requirements in the scientific/engineering computation domain was FORTRAN. With the evolution of other compiled imperative programming languages such as C, C++, ADA and Modula-3 during the last ten years, choices for programming languages have grown.
Modern programming languages enable to write programs, libraries and their interfaces in a clearer, more stringent way than a second generation language like FORTRAN 77. The use of a modern programming language should lead, at least in theory, to an increased productivity, namely because taking advantage of the language's facilities.

- it is easier to understand and use a library interface; furthermore, a well designed interface can prevent a lot of errors.
- structuring or modularizing the code inside a library helps the programmer to better understand the behavior of the program.

Sometimes, this increased productivity comes at a cost, however. Using high-level constructs that some modern programming languages advertise, can lead to a significant decrease in performance, if not used with care. The availability of compilers on different platforms which adhere to the same standard(s) and produce object code of good quality, can be a problem as well. Finally, connecting legacy software written in FORTRAN to the newly written software can be a problem.
B2000++ is a study of a C++ implementation of B2000. Its goals were:

- it should take advantage of C++'s features while retaining the same performance
- there should be the possibility of adding new element types without having to recompile or even rewrite B2000++.
- find out where C++ brings advantages compared to C and FORTRAN, and where there are problems with the use of C++
- implement it in such a way that a FORTRAN API would be possible
- use MemCom as persistent storage, so that both B2000and B2000++would be data-compatible
- the results of this study should provide information and guidelines as to how a successor of B2000 could be designed.

Why C++? There are other alternatives to choose from. A main reason for favoring C++ over languages such as ADA was the experience that we already had gathered. Another reason is that C++ is quite well accepted for numerical computations.

2 The Benefits and Disadvantages of Exceptions

Notice: Experienced C++ Programmers may skip this section.
Each programming language brings its own set of specifics as to how to program most effectively in that particular language. Below are a few listed that apply to C++:

- group data that belongs together into structures
- code that operates on particular data structures should be packed together with the data structures into classes
- where the set of algorithms can be applied to different types or classes, these algorithms should be written as template functions or as methods of a template class.
- use exceptions to control the program flow in case of errors.

The facilities mentioned above (structures, classes, templates and exceptions) should make life easier both for the implementors and the users of a software package. But many of the features that C++ offers cannot be used together with traditional coding style:

```
 1 void
 2 printfile(const char* filename)
 3 {
 4     FILE*   f;
 5     char*   line;
 6
 7     if ((f = fopen(filename, "r")) == NULL)
 8         return NULL;
 9     line = new char[1024];
10     while (fgets(line, sizeof(line), f) != NULL)
11         fprintf(stdout, "%s", line);
12     delete line;
13     fclose(f);
14 }
```

In this example, we have not really used any of the object-oriented elements of C++. Still, this code presents a serious problem: resource allocation together with exceptions. The operator new, which is used to allocate a string buffer, can raise an exception. In the case of an exception, the FILE structure allocated with fopen() does not get freed, because fclose() is never called.
The only way to deal with this is to introduce for each resource type an wrapper class. In this example, a File class could be implemented (C++'s iostreams library could be used alternatively):

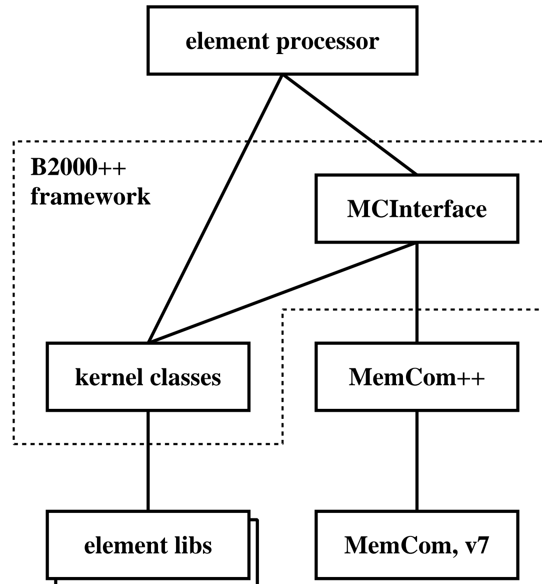
```
0  class File
1  {
2  private:
3      FILE* f;
4  public:
5      file(const char* name) {
6          if ((f = fopen(name, "r")) == NULL)
7              raise std::bad_alloc();
8      }
9     ~file() { fclose(f); }
10     FILE* operator*() { return f; }
11 };
```

For the the buffer we could create a Buffer class in a similar way. Writing a correct printfile function which does work in a context where exceptions are used, becomes much simpler:

```
1 void
2 printfile(const char* filename)
3 {
4     File    f(filename);
5     Buffer  line(1024);
6
7     while (fgets(line.data(), line.size(), *f) != NULL)
8        fprintf(stdout, "%s", line.data());
9 }
```

The price to pay for the convenience of exceptions is that functions calls which allocate and deallocate resources, must be encapsulated into classes so that the deallocation of the resource takes place automatically in the case of an exception. This is necessary for *every* type of resource, which makes it almost always necessary to write wrapper classes for every C- or `FORTRAN` library. This was also the case for the MemCom C API, where `MemCom++`, a `C++` wrapper, had to be implemented.
Concluding this section, one can say that `C++`'s paradigms in general do not allow free mixing of software written with in different programming styles; furthermore, having to wrap legacy code into classes adds bloat to the application. On the other hand, when writing code entirely in `C++`, one can greatly benefit from the use of exceptions.

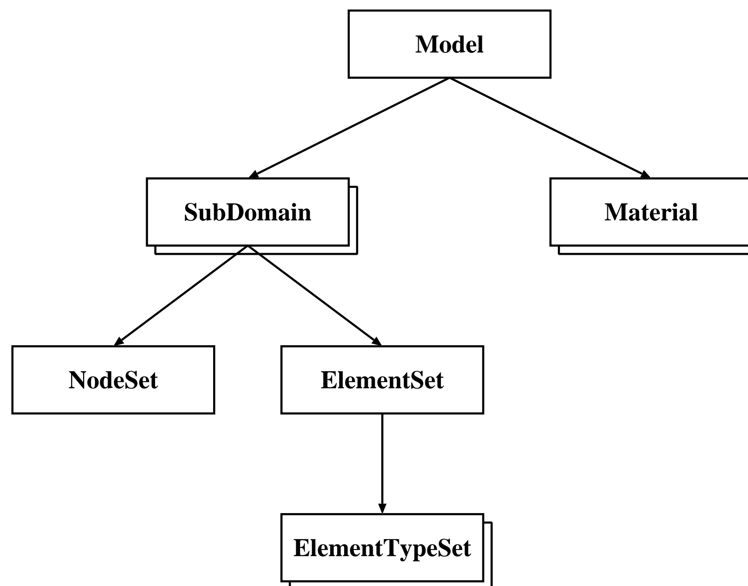
3 Overview of B2000++



B2000 uses MemCom for persistent storage of its data structures. For the reasons stated above, B2000++ uses MemCom++, the C++ wrapper of the MemCom API. The MCInterface class is a layer between MemCom++ and the element processor (which is not part of B2000++). MCInterface implements methods for storing and retrieving whole models and branches of a model. The kernel classes implement the data structures for the model. They provide fast and convenient methods for accessing model data.

4 Introduction to the Kernel Classes

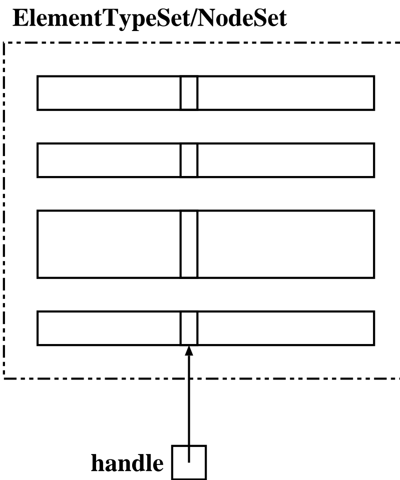
They are named this way because they form the core of every B2000 application. The following figure shows the objects that reference each other, forming a tree. At the root of the tree is the Model object, which has a reference to material data and to each Sub domain object in that model. A Sub domain object has references to a single NodeSet, which contains all nodes of that branch, and to a ElementSet object, which is a collection of ElementTypeSet objects, each of which contains the elements of a certain type.



This layout has several advantages:

- all objects in the tree can be accessed from the Model object. Thus it suffices to have a reference to the Model object.
- By grouping elements of the same type (which have therefore the same size) together, it is possible to use arrays to store the elements in. This makes the access of the elements simpler and more efficient than having elements of different storage sizes grouped together.

A requirement for `B2000++` was the possibility to write a `FORTRAN` API for it. Since `FORTRAN 77` does not support structured data, it was not possible to group all data belonging to an element into a C-struct or a `C++` class and then instantiate an array of that type. Instead, for every data item of an element, an array containing all these items for all elements had to be used. This requirement however would make the `C++` interface much more complicated and unintuitive than it needed to be.
Fortunately, `C++` provides operators and the possibility of in-lining of functions. This allows for a decoupling of interface and implementation, while retaining performance. In the case of `B2000++`, an element or a node is referenced by a handle. For the programmer, a handle is an objects which provides methods to access a given element/node, such as coordinates, etc. In reality, this handle is not much more but an index to the position of that element/node in the diverse arrays of an ElementTypeSet/NodeSet.



5 Adding Element Types

In order to be able to dynamically add new element types, an element library interface had to be implemented. Among other methods, the Sub Domain class provides the following methods, which are used by the element processor:

- `id computePreVar(Model* model);`
- `void computeVar1(Model* model);`
- `void computeVar2(Model* model);`
- `void computeVar3(Model* model);`
- `void computeMassMatrix(Model* model);`
- `void computeGradient(Model* model);`

These functions will call for each ElementTypeSet in the sub domain the corresponding elements library function (which has the same name). In order to dynamically add an element library, one must create a new object of class ElementTypSet by calling its constructor with the ElementTypeID of the desired element type:

```
ElementTypeSet::ElementTypeSet(const ElementTypeID type)
```

The constructor will then locate the element library, dynamically link it and check for the six functions figured above plus the initialize() function, which must also be in that library. It then calls this initialize() function, which returns the element type's parameters (ElementParam object). The newly created ElementTypeSet object can then be added to the sub domain's ElementSet, which is a collection of all ElementTypeSet objects of the sub domain.

6 The Helper Classes

The classes outlined above use a few helper classes. These are explained in this section.

6.1 ID's

To designate different Material-, SubDomain-, and ElementTypeSet objects in a mostly language-independent form (`FORTRAN 77` cannot handle pointers), unique identifiers are used in `B2000++`. There are also ID's for elements of an ElementTypeSet and ID's for nodes. Due to `FORTRAN 77`, these ID's are signed integers.

type name	C type	range of valid ID's	invalid ID
MaterialID	MCINT	0 to INT_MAX	-1
SubDomainID	MCINT	0 to INT_MAX	-1
NodeID	MCINT	0 to INT_MAX	-1
ElementID	MCINT	0 to INT_MAX	-1
ElementTypeID	MCINT		0, -1

6.2 ElementTypeID

The old `B2000` framework used strings as identifiers for element types (example: `HE20.ST`). Since strings are more difficult and more time consuming to handle than integer numbers, `B2000` internally used integer numbers and established a biunivocal relationship between string identifiers and integer identifiers by hardwiring the string and integer constants in the `B2000` program code. This approach causes problems when new element types are added: the string to integer type ID conversion routines must be updated.
In `B2000++`, this problem is eliminated by defining an algorithm which establishes the biunivocal relationship between string and integer ID's. An element type ID – whether represented by a string or by an integer – has four fields:

- the generic type
- the order
- "poly"
- the operator

Since any element type is defined by these four fields, only the conversions between string and integer representation for the fields must be hardwired.

6.3 ElementParam

For a given instance of ElementTypeID, the element parameters (such as the number of nodes per element) are fixed. Therefore there is one instance of ElementParam for each instance of ElementTypeSet. It is initialized by the initialize() function of the corresponding shared library.

6.4 Field

A Field is an array of doubles together with a factor, which tells how many doubles make up one value in the field. A scalar field would have a factor of 1, whereas a gradient field would have a factor of 3.

6.5 Container Classes

`B2000++`uses a number of specialized container classes. Most prominent are Array and FixedArray. Array behaves much like the STL vector class, except that it is only useful with primitive data types (not classes), but is slightly faster. FixedArray is like Array, except that resizing is not possible.

7 A detailed look at the ElementTypeSet Class

The most complicated of all classes in `B2000++` is the ElementTypeSet class. For a given sub domain, it holds all elements of a particular type. The members of this class are numerous. First come the accessor functions for the element's parameters. Where possible, the short name (due to `FORTRAN`) was replaced by a more meaningful name. These parameters are set by the element's library in the initialize() function and cannot be modified afterwards.

- `inline int numNodes() const;`
- `inline int numComputationalNodes() const;`
- `inline int lengthElementVector() const;`
- `inline int highestFreedomNumber() const;`
- `inline int lengthElementPreVariational() const;`
- `inline int lengthElementUpdatedRefFrame() const;`
- `inline int lengthGradient() const;`
- `inline int lengthPlasticity() const;`
- `inline int lere() const;`
- `inline int letp() const;`
- `inline int flgg() const;`

- inline int flgn() const;
- inline int flcs() const;
- inline int flcm() const;
- inline int flas() const;
- inline int numEdges() const;
- inline int numFaces() const;
- inline int numProperties() const;

All data is internally stored in arrays of primitive types, all arrays having the same size. The capacity indicates how many elements can be added without having to reallocate the data structure. Adding elements is done with the resize() function, which will allocate space for more elements. Afterwards these elements must be initialized and then validated with a call to validate(). To remove elements, the validate() function can be used to invalidate elements. When compacting a sub domain, these "holes" are removed from all the ElementTypeSets and the NodeSet (which is analogous).

- inline size_t size() const;
- inline bool empty() const;
- inline size_t capacity() const;
- void resize(size_t s);
- inline ElementTypeSet& validate(bool value = true);
- inline FixedArray<unsigned char> valid();
inline const FixedArray<unsigned char> valid() const;

An important concept in B2000++ is the iterator, which allows to access elements/node sequentially, much like a for loop with an index into an array. The advantage of iterators is that they hide such implementation details. The iterator is an iterator on handles, meaning that with a simple dereferencing the handle is obtained, from which all properties of an element can be accessed. The functions below return iterators to the first and the last element.

- inline iterator begin();
inline const iterator begin() const;
- inline const iterator end() const;
inline iterator end();

Instead of accessing the element's properties via iterators and handles, one can access them directly by getting the respective array. This is used for reading in and writing out data in one chunk by the MCInterface class and can also be used to create a `FORTRAN 77` API.

- `inline FixedArray<NodeID> connectivity();`
  `inline const FixedArray<NodeID> connectivity() const;`
- `inline FixedArray<double> properties();`
  `inline const FixedArray<double> properties() const;`
- `inline FixedArray<MaterialID> materials();`
  `inline const FixedArray<MaterialID> materials() const;`
- `inline FixedArray<IntegrationPoints> integrationPoints();`
  `inline const FixedArray<IntegrationPoints> integrationPoints() const;`
- `inline Array<Transform3D>& bases();`
  `inline const Array<Transform3D>& bases() const;`
- `inline FixedArray<Transform3DID> baseIDs();`
  `inline const FixedArray<Transform3DID> baseIDs() const;`
- `inline FixedArray<ExternalElementID> externalElementIDs();`
  `inline const FixedArray<ExternalElementID> externalElementIDs() const;`
- `inline pair<FixedArray<double>, bool> values(const FieldID i);`
  `inline const pair<FixedArray<double>, bool> values(const FieldID i) const;`
- `void clear();`

Finally, there is the possibility to add an element field. There is no limit to the number of fields. The field's values can be accessed by calling the values() function with the appropriate FieldID.

- `inline FieldID addField(const size_t numValues);`

The handle class, which is associated to the ElementTypeSet class, exports the following member functions:

- `inline ElementID& ID();`
  `inline ElementID ID() const;`

  Get the ElementID. The ElementID is the index of the Element in the arrays of the ElementTypeSet, ranging from 0 to `ElementTypeSet::size() - 1`.

- `inline FixedArray<NodeID> connectivity(); inline const FixedArray<NodeID> connectivity() const;`

  Get the connectivity of that Element.

- `inline FixedArray<double> properties(); inline const FixedArray<double> properties() const;`

  Get the properties of that Element.

- `inline MaterialID& material();`
  `inline MaterialID material() const;`

  get the Material ID of the Element. The material data of an Element can be accessed via

  ```
  Model m = ...
  ElementHandle e = ...

  Material *mat = m.material(e.material());
  ```

- `inline IntegrationPoints& integrationPoints();`
  `inline const IntegrationPoints& integrationPoints() const;`

  Get the number of integration points in each direction for that Element.

- `inline FixedArray<double> value();`
  `inline const FixedArray<double> value() const;`

  Get the value for the currently selected Field.

- `inline bool& valid(); inline bool valid() const;`

  Returns true if it is a valid Element. This function assumes that the ElementHandle points to an existing SubDomain and to an existing ElementTypeSet.

- `inline ExternalElementID externalElementID() const;`
  `inline ExternalElementID& externalElementID();`

- `inline void selectField(const FieldID i);`

  Select the current Field for that Element. This always has to be set (of course with a valid FieldID for a field which has been added to the ElementSet) before calling `value()`.

- `inline handle& operator=(const handle& e);`

  assigns the content of another Element (pointed to by `e`) to this Element.

8 Storage

In order to store or retrieve models and sub domains, the MCInterface class can be used. The implementation of this class was quite difficult, because it was required to be compatible with the `B2000` layout of MemCom databases. Among the missing things are methods to load/save parts of a Model according to the current computation stage of the element processor.

9 Implementation

A demo program which uses the B2000++ library, was implemented. It loads a sub domain from a database and then iterates through all quad elements, using iterators. The execution time was compared with a C-only version (using MemCom-7).

10 Performance

The loading of a sub domain is much faster with B2000++than with B2000 version 2.3. This is due to an ETAB-related optimization in the MCInterface class, but not relevant for the assessment of the C++ aspects (iterators). When compiling with gcc, a severe performance hit was noticed when iterating over the elements. This is due to gcc's current inability to properly resolve variables and changes to them in nested inline functions. The performance penalty ranged up to a factor of 30 with gcc. With a bit of tweaking in the source code, this could be reduced to a factor of two. But this was still inacceptable for numerical computing.
Therefore, another compiler was tried: KAI C++, which is a C++ to C compiler that makes use of gcc as back end on x86 machines. Therefore, it is well suited for a comparison with the plain C program. The results showed no difference between the B2000++- and the C-only version, indicating that KAI C++ does a much better job optimizing C++ code than gcc. However, KAI C++ is not free software.

11 Conclusion

Nearly all of the goals of this study have been achieved. B2000++ is easier to use, more flexible and extendible than B2000, while providing the same performance characteristics and the possibility for backward compatibility. The disadvantages are that one must use a commercial compiler to achieve acceptable performance and that C++ code using exceptions does not mix well with other code.
B2000++ incorporates a number of new ideas and concepts, which can be taken into consideration in the analysis and design phase of a successor for B2000.