FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

A C++ Vector Container for Pointers to Polymorphic Objects

Klaus Iglberger, Ulrich Rüde

Technical Report 09-11

A C++ Vector Container for Pointers to Polymorphic Objects

Klaus Iglberger, Ulrich Rüde

Lehrstuhl für Systemsimulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

klaus.iglberger@informatik.uni-erlangen.de

July 6, 2009

Depending on the application, storing pointers in a container can be associated with several drawbacks concerning the semantics of element access and resource management. In this technical report, we introduce the `PtrVector`, a resource managing vector container for polymorphic pointers. In contrast to `std::vector` from the standard library and `ptr_vector` from the Boost library, it provides configurable management of the stored resources, configurable growth, and implements the traversal of subsets of contained polymorphic pointers.

1 Motivation

The `std::vector` is one of the standard libraries most useful tools [LL98]. It is the standard solution for a dynamically allocated, automatically growing, and memory managed array. It provides fast random access to its elements, since a vector guarantees that the elements lie adjacent in memory and manages the dynamically allocated memory according to the RAII [1] idiom. Yet there are some situations, where users of `std::vector` experience several drawbacks, especially when `std::vector` is used in combination with pointers. For instance, a `const_iterator` over a range of pointer will not allow the stored pointers to change, but the objects behind the pointers remain changeable. The following example illustrates that it is possible to change the values of `double`s through an iterator-to-const:

Listing 1: Behavior of a std::vector for pointers

```
1  typedef std::vector<double*>  Doubles;
2
3  Doubles doubles;  // Creating a vector for pointers to double values
4
5  // Filling the vector with pointers to double values. All values are initialized with 1.
6  for( size_t i=0; i<10; ++i )
7     doubles.push_back( new double( 1.0 ) );
8
9  // Accessing the first rigid body
10 Doubles::const_iterator first = doubles.begin();
11 **first = 2.0;  // Changes the double value through an iterator-to-const
```

The basic reason for this behavior is that `std::vector` is unaware of the fact that it stores pointers instead of objects and therefore the pointer are considered constant, not the objects behind the pointer.

Another drawback of `std::vector` is the fact that during destruction of a vector object the dynamically allocated bodies are not deleted. Again, `std::vector` is unaware of the special property of pointers and

[1] Resource Acquisition Is Initialization

therefore does not apply any kind of deletion policy. It basically calls the default destructor for pointers, which in turn does nothing and especially does not destroy the attached objects. [2]

Since `std::vector` is not a good choice for the use with pointers, the Boost library [Boo, Kar08] provides a special vector container for pointers, `ptr_vector`. In contrast to `std::vector`, the Boost `ptr_vector` explicitly knows that it deals with pointers to objects and therefore automatically takes care of the management of these dynamically allocated objects. It also creates the appearance that the contained elements are objects instead of pointers, which improves the user interface to the contained pointers:

Listing 2: boost::ptr_vector Example

```
typedef boost::ptr_vector<double>  Doubles;
Doubles doubles;  // Creating an empty ptr_vector for pointers to double values

doubles.push_back( new double(1.0) ); // A new pointer-to-double is added to the vector
doubles[0] = 2.0;                     // ... but used as if it was an object

Doubles::iterator first = doubles.begin();
*first = 3.0;  // No indirection needed

Doubles::const_iterator second( first+1 );
*second = 4.0;  // Compile time error! It is not possible to change double
                // values via an iterator-to-const
```

However, unfortunately even the Boost `ptr_vector` lacks some functionality desirable in certain situations. One necessary feature is the control over the deletion behavior of the pointer vector. Per default, the Boost `ptr_vector` applies operator `delete` to all pointers upon destruction. It is not possible to change this behavior, for instance to either switch off the destruction or to adjust to pointers to arrays.

Additionally, it is not easily possible to return the dynamic type of the contained objects. This is arguably in general not a very important feature, however for specific applications it might be. In order to demonstrate this requirement, let's consider the following class hierarchy [3]:

Listing 3: Rigid body class hierarchy

```
class RigidBody { /* ... */ };

class Sphere : public RigidBody { /* ... */ };

class Box : public RigidBody { /* ... */ };

class Cylinder : public RigidBody { /* ... */ };
```

Now assume that we have a vector full of pointers to rigid bodies and we are in need to know exactly what geometry these bodies have in order to perform collision tests between the rigid bodies. Wouldn't the following code example make everything very easy for us?

Listing 4: Selective traversal of a pointer vector

```
typedef PtrVector<RigidBody>         BodyVector;
typedef BodyVector::SphereIterator   SphereIterator;

// Iterating over all spheres contained in the vector of rigid body pointers
for( SphereIterator sphere=bodies.beginSpheres();
     sphere!=bodies.endSpheres();
     ++sphere )
{
   ... // Using some special properties of a sphere not accessible
       // through the pointer to the RigidBody base class
}
```

In general, using pointers to rigid bodies to abstract from the actual geometry of the rigid bodies is the usual way to handle all rigid bodies equally, but in some cases we are in dire need to know some special properties of the bodies, as for example their geometry, the according expansions, etc.

Due to the deficiencies of both `std::vector` and the Boost `ptr_vector`, we introduce the `PtrVector` class of the *pe* physics engine. It offers the same access semantics as the Boost `ptr_vector`, i.e. removes the need for an extra indirection, but offers the possibility to configure the memory management and the

[2] Note, that this behavior could be alleviated by a veneer class (see [Wil05]), that has a destructor destroying all dynamically allocated objects. Thus in case this is the only problem, it can be solved easily.

[3] This hierarchy is the simplified class hierarchy of the *pe* physics engine.

growth of the internal storage, and implements special features for polymorphic pointers, as for instance a convenient way to iterate over a subset of polymorphic objects contained in the pointer vector.

2 Implementation of the Pointer Vector

The following code shows the class declarations from the `std::vector`, `boost::ptr_vector`, and `pe::PtrVector` classes:

Listing 5: Declarations of the std::vector, boost::ptr_vector, and pe::PtrVector classes

```
// Declaration of the std::vector class
template< typename _Tp       // Type of the contained elements
        , typename _Alloc >  // Type of the used allocator
class vector;

// Declaration of the boost::ptr_vector class
template < class T               // Type of the contained pointers
         , class CloneAllocator  // Allocator for the cloning process
         , class Allocator >     // Type of the used allocator
class ptr_vector;

// Declaration of the pe::PtrVector class
template< typename T    // Type of the contained pointers
        , template D    // Deletion policy
        , typename G    // Growth policy
        , typename A >  // Allocator type
class PtrVector;
```

The declarations of the classes look very similar. However, there are several differences to note. In contrast to the Boost `ptr_vector` class, the *pe* `PtrVector` does not offer any `CloneAllocator`. Instead, `PtrVector` offers the configuration of its behavior via two different template argument: the deletion and the growth policy.

The deletion policy adds the flexibility to `PtrVector` to configure the memory management behavior. Each time a pointer is erased from the vector, the deletion policy is applied to free the corresponding resources. For instance, in case the `PtrVector` is supposed to destroy all contained rigid bodies, the `PtrDelete` policy class can be used:

Listing 6: Application of the pe::PtrDelete policy

```
struct PtrDelete
{
   template< typename Type >
   inline void operator()( Type ptr ) const
   {
      delete ptr;
   }
};

typedef pe::PtrVector<RigidBody,PtrDelete>  Bodies;
Bodies bodies;
bodies.pushBack( createSphere( ... ) );  // Add a new sphere to the pointer vector
bodies.pushBack( createBox   ( ... ) );  // Add a new box to the pointer vector

// Automatic destruction of the two contained rigid bodies
```

Note two details in this example: the `PtrDelete` class itself is not a template, but its contained function call operator, which makes the class applicable to any kind of pointer, and the slightly different naming convention of *pe* in comparison to the standard library and Boost (`pushBack` instead of `push_back`) [SA08].

In case the `PtrVector` should not take responsibility for the contained resources, the `NoDelete` policy class can be used:

Listing 7: Application of the pe::NoDelete policy

```
struct NoDelete
{
   template< typename Type >
   inline void operator()( const Type& ptr ) const
   {}
};

```

```
typedef pe::PtrVector<RigidBody,NoDelete>  Bodies;
Bodies bodies;
bodies.pushBack( createSphere( ... ) );  // Add a new sphere to the pointer vector
bodies.pushBack( createBox   ( ... ) );  // Add a new box to the pointer vector

// No destruction of the two contained rigid bodies, memory must be managed elsewhere
```

Again note the fact, that not the NoDelete class is templatized, but the function call operator, which makes the class universally useable.

The growth policy enables the user to specify how a pointer vector grows in case it needs more elements. Although in most cases the optimal growth strategy suggested by Andrew Koenig [Koe98, Sut07] should provide the best performance for most scenarios, in some scenarios a different approach can still make a difference. Example 8 illustrates the `OptimalGrowth` policy class:

Listing 8: Implementation of the pe::OptimalGrowth policy

```
struct OptimalGrowth
{
   inline size_t operator()( size_t oldSize, size_t minSize ) const
   {
      const size_t needed( max( static_cast<size_t>( old*1.5 ), minimum ) );
      return ( ( needed )?( 4 * ( (needed-1)/4 + 1 ) ):( 0 ) );
   }
};
```

Additionally, we adopt two very useful strategies for our pointer vector. First, we consider the objects behind a `const`-iterator as constant (not only the pointer) and second we don't allow the change of the container via the iterators, i.e. in order to add, change or erase an element of the container, we need the container itself, not only an iterator over its elements. This is very conveniently for our engine, because with this convention we are able to give a user iterators over the range of our internally stored elements without fear that the user might change the internal storage:

Listing 9: Iterator behavior of the pe::PtrVector class

```
// Getting a handle to the rigid body simulation world
WorldID world = theWorld();

// Iterating over the rigid bodies contained in the
// simulation world. The user has only access to the
// rigid bodies via the pair of iterators and can
// therefore not change the internal storage of the
// simulation world.
for( World::Iterator body=world->begin();
     body!=world->end();
     ++body )
{
   body->...;       // Doing something with the rigid body
   *body = ...;     // Compile time error! It is not possible to change the pointer!
   body[0] = ...;   // Compile time error! No subscript operator available!
}
```

The code listing 10 shows the entire public interface of the `PtrVector` class. Note that it is possible to distinguish between `erase`, which erases an element from the pointer vector by applying the deletion policy, and `release`, which releases an element from the pointer vector without applying the deletion policy. Also note the templated versions of the `size`, `begin`, and `end` functions, which will be explained in detail in the next section.

Listing 10: Class definition of the PtrVector class

```
template< typename T                    // Type
        , typename D = PtrDelete        // Deletion policy
        , typename G = OptimalGrowth >  // Growth policy
class PtrVector
{
 public:
   // ...

   //**Constructors*****************************************************************
   explicit inline PtrVector( SizeType initCapacity = 0 );
            inline PtrVector( const PtrVector& pv );

```

```
13     template< typename T2, typename D2, typename G2 >
14              inline PtrVector( const PtrVector<T2,D2,G2>& pv );
15     //**********************************************************************************
16
17     //**Destructor*********************************************************************
18     inline ~PtrVector();
19     //**********************************************************************************
20
21     //**Assignment operators***********************************************************
22     PtrVector& operator=( const PtrVector& pv );
23
24     template< typename T2, typename D2, typename G2 >
25     PtrVector& operator=( const PtrVector<T2,D2,G2>& pv );
26     //**********************************************************************************
27
28     //**Get functions******************************************************************
29                            inline SizeType maxSize()  const;
30                            inline SizeType size()     const;
31     template< typename C > inline SizeType size()     const;
32                            inline SizeType capacity() const;
33                            inline bool     isEmpty()  const;
34     //**********************************************************************************
35
36     //**Access functions***************************************************************
37     inline PointerType      operator[]( SizeType index );
38     inline ConstPointerType operator[]( SizeType index ) const;
39     inline PointerType      front();
40     inline ConstPointerType front() const;
41     inline PointerType      back();
42     inline ConstPointerType back()  const;
43     //**********************************************************************************
44
45     //**Iterator functions*************************************************************
46                            inline Iterator             begin();
47                            inline ConstIterator        begin() const;
48     template< typename C > inline CastIterator<C>      begin();
49     template< typename C > inline ConstCastIterator<C> begin() const;
50
51                            inline Iterator             end();
52                            inline ConstIterator        end()   const;
53     template< typename C > inline CastIterator<C>      end();
54     template< typename C > inline ConstCastIterator<C> end()   const;
55     //**********************************************************************************
56
57     //**Element functions**************************************************************
58     inline void     pushBack   ( PointerType p );
59     inline void     popBack    ();
60     inline void     releaseBack();
61
62     template< typename IteratorType >
63     inline void     assign( IteratorType first, IteratorType last );
64
65     inline Iterator insert( Iterator pos, PointerType p );
66
67     template< typename IteratorType >
68     inline void     insert( Iterator pos, IteratorType first, IteratorType last );
69
70     inline Iterator erase  ( Iterator pos );
71     inline Iterator release( Iterator pos );
72     inline void     clear  ();
73     //**********************************************************************************
74
75     //**Utility functions**************************************************************
76            void reserve( SizeType newCapacity );
77     inline void swap( PtrVector& pv ) throw();
78     //**********************************************************************************
79
80     // ...
81 };
```

3 Implementation of a Vector for Polymorphic Pointers

The second desired feature is a selective traversal of our pointer vector over all elements of a particular dynamic type. It should for example be possible to traverse all rigid spheres in the vector, although it could be possible that it additionally stores several rigid boxes, cylinders, or other body types. Additionally it should be possible to count the number of polymorphic objects of a particular dynamic type contained in the pointer vector:

Listing 11: Implementation goal for a vector for polymorphic pointers

```
typedef PtrVector<RigidBody>         BodyVector;
typedef BodyVector::SphereIterator   SphereIterator;

BodyVector bodies;

// ... Filling the vector with rigid bodies of different geometries ...

// Iterating over all spheres contained in the vector of rigid body pointers
for( SphereIterator sphere=bodies.beginSpheres();
     sphere!=bodies.endSpheres();
     ++sphere )
{
   ... // Using some special properties of a sphere not accessible
       // through the pointer to the RigidBody base class
}

// Calculating the total number of boxes contained in the vector
std::size_t numBoxes = bodies.countBoxes();
```

The first thing to realize is the fact that it is not possible to implement a general solution for this problem by using named functions to do the job (as for instance `beginSpheres()`. We definitively have to use a templated approach. It is therefore necessary to add a templated `begin`, `end`, and `size` function to the pointer vector as illustrated in listing 10. Consequently, it is also necessary to have a templated `Iterator` available that gets the desired dynamic type as template argument:

Listing 12: Templated begin, end, and size functions

```
typedef PtrVector<RigidBody>          BodyVector;
typedef BodyVector::Iterator<Sphere>  SphereIterator;

BodyVector bodies;

// ... Filling the vector with rigid bodies of different geometries ...

// Iterating over all spheres contained in the vector of rigid body pointers
for( SphereIterator sphere=bodies.begin<Sphere>();
     sphere!=bodies.end<Sphere>();
     ++sphere )
{
   ... // Using some special properties of a sphere not accessible
       // through the pointer to the RigidBody base class
}

// Calculating the total number of boxes contained in the vector
std::size_t numBoxes = bodies.size<Sphere>();
```

However, this will unfortunately result in a naming conflict between the default `Iterator` over all contained vector elements and the templated iterator since we cannot use the name `Iterator` both with and without a template argument (as `Iterator` for an iterator over all rigid bodies and for instance `Iterator<Sphere>` for an iterator over all contained spheres). Therefore we will use the name `CastIterator` instead:

Listing 13: Templated begin(), end(), and size() functions

```
typedef PtrVector<RigidBody>              BodyVector;
typedef BodyVector::CastIterator<Sphere>  SphereIterator;

// ...
```

Although this code example promises an immediate, intuitive understanding of the code's effect, it also raises the question of a possible implementation of `CastIterator`. A first implementation approach unfortunately fails due to some limitations of C++:

Listing 14: First implementation approach for CastIterator

```
template< typename D    // Desired dynamic type of
                        // the elements
        , typename S >  // Static type of the
                        // underlying elements
class CastIterator
{
   ...
};

template< typename T                    // Type
        , typename D = PtrDelete        // Deletion policy
        , typename G = OptimalGrowth >  // Growth policy
class PtrVector
{
 public:
   typedef ...  // How to do the typedef ??
}
```

In case the `CastIterator` was implemented as a separate class template, it would not be possible [4] to create a `typedef` for the `CastIterator` inside the `PtrVector` class that would enable the following syntax:

Listing 15: Desired CastIterator syntax

```
PtrVector<RigidBody>::CastIterator<Sphere> ...
```

One solution in this situation could be a nested class inside the `PtrVector` class deriving from the `CastIterator` class. However, an even simpler approach is to directly implement the `CastIterator` as nested class inside the `PtrVector` class template:

Listing 16: Desired CastIterator syntax

```
template< typename T                    // Type
        , typename D = PtrDelete        // Deletion policy
        , typename G = OptimalGrowth >  // Growth policy
class PtrVector
{
 public:
   template< typename C >  // Desired dynamic type of the elements
   class CastIterator;
}
```

By implementing the `CastIterator` as a nested class inside `PtrVector` it is no longer necessary to create a `typedef` for a class template. Additionally, we save the second template parameter for the `CastIterator` since the static type of the elements is already known (the type `T` of the `PtrVector` class template.

The following listing gives an impression of the most important parts of a first implementation of the `CastIterator` class template. In this listing, note the systematic application of compile time constraints to check whether the given template arguments are correct. For more informations on the *pe* compile time constraints, refer to [IR09], where you will find a detailed introduction to the purpose and implementation of compile time constraints.

Listing 17: Initial implementation of the nested CastIterator class template

```
template< typename T    // Type
        , typename D    // Deletion policy
        , typename G >  // Growth policy
template< typename C >  // Cast type
class PtrVector<T,D,G>::CastIterator
{
 private:
   IteratorType cur_;  // Pointer to the current memory location.
   IteratorType end_;  // Pointer to the element one past the last element in the element range.

 public:
   ... // The necessary type definitions

   // Default constructor for CastIterator
```

[4] Yet? Suggestions like this one have been discussed for the next C++ standards.

```
15    inline CastIterator()
16       : cur_(NULL)
17       , end_(NULL)
18    {
19       pe_CONSTRAINT_MUST_BE_STRICTLY_DERIVED_FROM( C, T );
20    }
21
22    // Standard constructor for CastIterator
23    inline CastIterator( IteratorType begin, IteratorType end )
24       : cur_(begin)
25       , end_(end  )
26    {
27       pe_CONSTRAINT_MUST_BE_STRICTLY_DERIVED_FROM( C, T );
28       while( cur_ != end_ && !dynamic_cast<C*>( *cur_ ) ) ++cur_;
29    }
30
31    // Copy constructor for other instances of the CastIterator
32    template< typename Other >
33    inline CastIterator( const CastIterator<Other>& it )
34       : cur_( it.base() )
35       , end_( it.stop() )
36    {
37       pe_CONSTRAINT_MUST_BE_STRICTLY_DERIVED_FROM( C, T );
38       pe_CONSTRAINT_POINTER_MUST_BE_CONVERTIBLE( Other, C );
39    }
40
41    // Prefix increment operator
42    inline CastIterator& operator++()
43    {
44       while( ++cur_ != end_ && !dynamic_cast<C*>( *cur_ ) ) {}
45       return *this;
46    }
47
48    // Postfix increment operator
49    inline CastIterator  operator++( int )
50    {
51       CastIterator tmp( *this );
52       while( ++cur_ != end_ && !dynamic_cast<C*>( *cur_ ) ) {}
53       return tmp;
54    }
55
56    // Dereference operator
57    inline PointerType operator*()  const
58    {
59       return static_cast<C*>( *cur_ );
60    }
61
62    // Member access operator
63    inline PointerType operator->() const
64    {
65       return static_cast<C*>( *cur_ );
66    }
67
68    // Access to the current element of the CastIterator
69    inline const IteratorType& base() const
70    {
71       return cur_;
72    }
73
74    // Access to the final element of the CastIterator
75    inline const IteratorType& stop() const
76    {
77       return end_;
78    }
79 };
```

The first apparent difference to other vector iterators is the fact that the `CastIterator` takes two arguments: one for the beginning and one for the end of the element range. The second argument is important since internally the `CastIterator` is moving through the elements until an element of dynamic type `C` is found. However, the search is ended in case the element one after the last element of the element range is encountered.

There are three functions, where it is necessary to search for the next fitting element: the standard constructor, the prefix increment operator and the postfix increment operator. In all three cases the iterator has to be moved to the next element of dynamic type `C`, which involves an element wise search of the element range until (at last) the element one past the last element of the element range is found. For the check whether or not an element has the desired dynamic type we could use a `dynamic_cast`, which is the most general and only portable approach for this problem. Due to the applied compile time constraints the cast from the static type `S` to the dynamic type `D` is safe.

Inside the two access operators of `CastIterator` only `static_cast`s are used. Since it has already been checked that the element behind the current element is of the dynamic type `C` it is safe to use the faster `static_cast`.

4 Operators for the CastIterator

The last missing piece for a working implementation are the comparison operators for the `CastIterator`:

```
typedef PtrVector<RigidBody>              BodyVector;
typedef BodyVector::CastIterator<Sphere>  SphereIterator;

for( SphereIterator sphere=bodies.begin<Sphere>;
     sphere!=sphere.end<Sphere>();
     ++sphere )
{
   ...
}
```

The implementation of these operators deserves some special treatment. According to the C++ standard (14.8.2.4/4) [Str03] it is not possible to deduce the template arguments of nested templates:

Listing 18: Global or namespace formulation of the CastIterator operators

```
template< typename T    // Type
        , typename D    // Deletion policy
        , template G >  // Growth policy
template< typename C >  // Cast type
bool operator==( typename PtrVector<T,D,G>::CastIterator<C>& lhs,
                 typename PtrVector<T,D,G>::CastIterator<C>& rhs )
{
   // ...
}
```

Although the formulation as global or namespace function is possible, the compiler will not be able to select this function for the comparison of two `CastIterator`s. The solution to this problem is the application of the old but still famous Barton-Nackman trick. Originally this trick was published to enable the overloading of operators in pre-standard C++, where overloading of template operators was not possible and namespaces were yet to come. By using this trick, it is possible to use the `CastIterator` as intended:

Listing 19: Barton-Nackman formulation of the comparison operators

```
template< typename T    // Type
        , typename D    // Deletion policy
        , template G >  // Growth policy
class PtrVector
{
 public:
   ...
   // Declaration of the nested CastIterator
   template< typename C > CastIterator;

   // In-class implementation of the comparison operator for two cast iterators
   template< typename L, typename R >
   friend inline bool operator==( const CastIterator<L>& lhs,
                                  const CastIterator<R>& rhs )
   {
      return lhs.base() == rhs.base();
   }

   // ... Same for all other necessary comparison operators
}
```

Every time a `PtrVector` gets instantiated, the comparison operators defined inside the class are injected into the surrounding namespace (in this case the `pe` namespace). Arguably, depending on whether the compiler adds these `inline` functions to the object files this might increases the size of our executable, but this is (to our current knowledge) the only way to solve this problem. Now every time the compiler encounters a comparison between two `CastIterators` it can select one of the available ones since only the template arguments of the nested class are unknown (the template arguments of the `PtrVector` class template are fixed).

5 Specialization of the CastIterator

The result is a general solution for the requirement for an iterator over a specific subset of the elements of a certain dynamic type contained in a pointer vector. However, a small flaw remains in the design that needs to be addressed [5]. There may be cases where we can improve the performance for the `CastIterator` by skipping the comparatively expensive `dynamic_cast` to find out if an element has the desired dynamic type. One of these cases is the *pe* physics engine, where the type of the rigid bodies is encoded in the RigidBody base class [6]:

Listing 20: getType() function of the RigidBody class

```
1 class RigidBody
2 {
3    // ...
4    inline GeomType getType() const;
5    // ...
6 };
```

Due to this formulation, we can replace the `dynamic_cast` by a simple `if` statement.

The most obvious choice to adapt the behavior of `CastIterator` is a specialization of the according `CastIterator` functions (namely the standard constructor and the two increment operators). However, according to the C++ standard [Str03] it is only possible to specialize nested classes in case the surrounding class template has been specialized too [VJ03]. Therefore we would only be able to specialize the `CastIterator` functions for a specific `PtrVector` instantiation. This would definitively loose the flexibility that was added by introducing the policy template parameters for the `PtrVector` class template: it would be necessary to specialize the `CastIterator` for every required instantiation of the `PtrVector` class template for different growth and deletion policies.

This problem can be solved by an additional layer of indirection. Instead of implementing the search for a fitting element directly in the `CastIterator` functions, we introduce the generalized `polymorphicFind()` shim function:

Listing 21: Implementation of the polymorphicfind() shim function

```
1 template< typename D    // Dynamic type of the objects
2         , typename S >  // Static type of the objects
3 inline S *const * polymorphicFind( S *const * first, S *const * last )
4 {
5    pe_CONSTRAINT_MUST_BE_STRICTLY_DERIVED_FROM( D, S );
6
7    while( first != last && !dynamic_cast<D*>( *first ) ) ++first;
8    return first;
9 }
```

This function traverses the range $[first, last)$ of pointers to objects with static type `S` until it finds the next polymorphic pointer to an object of dynamic type `D`. Since this function now deals with the search for the next element with dynamic type `D` in the given element range, it is possible to implement the `CastIterator` functions in terms of this function:

Listing 22: Implementation of the polymorphicfind() shim function

```
1 template< typename T    // Type
2         , typename D    // Deletion policy
```

[5] "needs" to because we are perfectionists, aren't we?

[6] This solution is a compromise to improve the performance of one of the most important performance bottlenecks: the collision detection. The type of a rigid body is encoded as a constant integral in the rigid Body base class although there is good reason to argue against such a coding style (for a complete discussion see [Mey08] and [Dew07]). However, in *pe* rigid bodies are neither copyable nor cloneable, which removes one of the most severe drawbacks of such a programming style.

```
3          , template G >  // Growth policy
4  template< typename C >  // Cast type
5  class PtrVector<T,D,G>::CastIterator
6  {
7     // ...
8   public:
9     // ...
10
11    // Standard constructor for CastIterator
12    inline CastIterator( IteratorType begin, IteratorType end )
13       : cur_(begin)
14       , end_(end  )
15    {
16       cur_ = polymorphicFind<C>( cur_, end_ );
17    }
18
19    // ...
20
21    // Prefix increment operator
22    inline CastIterator& operator++()
23    {
24       cur_ = polymorphicFind<C>( ++cur_, end_ );
25       return *this;
26    }
27
28    // Postfix increment operator
29    inline CastIterator  operator++( int )
30    {
31       CastIterator tmp( *this );
32       cur_ = polymorphicFind<C>( ++cur_, end_ );
33       return tmp;
34    }
35
36    // ...
37 };
```

This design new enables the implementation of specialized versions of the `polymorphicFind` function that are used in every possible implementation of the `CastIterator` class template.

6 Conclusion

In this report we have presented an implementation for a C++ vector for polymorphic pointers. The `PtrVector` class template enables a convenient way to access the contained elements, offers the possibility to configure the deletion policy, and implements a scheme to traverse a subset of polymorphic objects of a specific dynamic type. Additionally it is possible to specialize this behavior in order to improve the performance for certain types. The implementation features demonstrated for a vector container can also be applied to other container types, as for instance lists, sets, or maps.

References

[Boo] Boost, *Homepage of the Boost C++ framework: http://www.boost.org.*

[Dew07] S.C. Dewhurst, *C++ Gotchas*, Addison-Wesley Professional Computing Series, Addison-Wesley, 2007, 14 GI mat 12.1.2.2405b.

[IR09] K. Iglberger and U. Rüde, *C++ Compile Time Constraints*, Tech. report, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2009.

[Kar08] B. Karlsson, *Beyond the C++ Standard Library. An Introduction to Boost*, Addison-Wesley, 2008, 14 GI mat 12.1.2-2530.

[Koe98] A. Koenig, *Why Are Vectors Efficient?*, JOOP **11** (1998), no. 5, 71–75.

[LL98] S.B. Lippman and J. Lajoie, *C++ Primer*, Addison-Wesley, 1998, ISBN 0-201-82470-1.

[Mey08] S. Meyers, *More Effective C++*, Addison-Wesley Professional Computing Series, Addison-Wesley, 2008, 14 GI mat 12.1.2.1797a.

[SA08] H. Sutter and A. Alexandrescu, *C++ Coding Standards*, C++ In-Depth Series, Addison-Wesley, 2008, 14 GI mat 12.1.2-2435.

[Str03] B. Stroustrup, *The C++ Standard Incorporating Technical Corrigendum 1*, second ed., Wiley, 2003.

[Sut07] H. Sutter, *More Exceptional C++*, C++ In-Depth Series, Addison/Wesley, 2007, 14 GI mat 12.1.2-2520.

[VJ03] D. Vandevoorde and N.M. Josuttis, *C++ Templates - The Complete Guide*, Addison-Wesley, 2003.

[Wil05] M. Wilson, *Imperfect C++*, Addison-Wesley, 2005, 14 Gi mat 12.1.2-2523.