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

Lehrstuhl für Informatik 10 (Systemsimulation)

C++ Compile Time Constraints

Klaus Iglberger, Ulrich Rüde

Technical Report 09-10

C++ Compile Time Constraints

Klaus Iglberger, Ulrich Rüde

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

klaus.iglberger@informatik.uni-erlangen.de

August 18, 2009

In this report we introduce the compile time constraint library of the *pe* physics engine. These compile time constraints are a valuable tool to detect compile time errors and in case of an error to abort the compilation process and report the error by emitting a comprehensible error message. Additionally, they offer the option to enforce certain design decisions.

1 Motivation

Most errors of a C++ program occur during the runtime of the program: invalid access indices, violated pre- and postconditions of a function, failed memory allocations, files that cannot be opened, to name a few. These kind of errors are usually handled appropriately by either throwing an exception or aborting the program by an assertion. However, some errors can already or only be detected at compile time. Using exceptions or assertions for these kind of errors would delay the error report until the program is run for the first time. Consider for example Listing 1:

Listing 1: First example for a compile time error

```
1 const std::size_t maxThreads = 4;
2 // ...
3 assert( maxThreads > 0 );  // Is the number of threads larger than zero?
```

Imagine that the maximum number of threads is configured via a constant integral value. However, at some point we have to make sure that this value is larger than zero (because zero threads would not be able to perform any work). Due to the use of an assertion, we are only able to detect the error at runtime, although the value is a compile time constant expression and could therefore be reported at compile time[1]. A different kind of error is illustrated in Listing 2:

Listing 2: Second example for a compile time error

```
1 typedef int  real;  // Is this really a floating point data type?
```

In this case we want to create a convenient switch between single-precision floating point values and double-precision floating point values by providing a single typedef defining the new floating point type `real`. This new type is now used throughout our program, which allows us to switch between `float` and `double` values by changing the type in the type definition. However, inadvertently the intended floating point type was set to an integral data type, which will probably make our results useless.

In order to detect errors of these kinds and to report them at compile time, it is desirable to add code that either aborts compilation and reports the error as descriptive as possible, or in case no error

[1] Even worse, we are only able to detect this error in case we did not specify NDEBUG during the compilation!

is detected cleanly compiles and creates no additional overhead at runtime. A solution for this problem are static assertions and compile time constraints, as for example proposed and implemented in the Boost library [Boo] and admitted to the technical report 1 (TR1) of the C++ standard.

In this report we introduce the compile time constraints of the *pe* physics engine [IR09]. They are a combination of the static assertion from the Boost library [Boo] and several ideas from Andrei Alexandrescu [Ale01] and Matthew Wilson [Wil05] to create comprehensible error messages. Section 2 will demonstrate several ways how to abort compilation in case of an error depending on a compile time constant expression and will introduce the Boost static assertion. Section 3 will handle the topic of comprehensible error message, before Section 4 will describe the *pe* compile time constraints in detail. Section 5 and Section 6 will highlight two interesting compile time constraints, before Section 7 will conclude the report.

2 Static Assertion

The basic idea of static asserts is to provide compile time validation of an expression. Clearly only compile time constant expressions can be evaluated at compile time:

Listing 3: Proper use of static assertions

```
1 // Compile time evaluation of a compile time static expression
2 STATIC_ASSERT( sizeof(double) >= sizeof(float) );
3
4 // Compilation error: attempt to evaluate a runtime expression at compile time
5 double& Vector::operator[]( size_t index )
6 {
7    STATIC_ASSERT( index < size_ );  // Bound check of the vector access index
8    // ...
9 }
```

The first of these two examples illustrates the use of a static assertion for a compile time constant expression, i.e., an expression whose result can be calculated at compile time. In this case the static assertion will not abort the compilation process, since the size of a double precision floating point value should always be larger or equal to a single precision floating point value. The second of the examples attempts to apply a static assertion to a runtime expression. Obviously, it is not possible to tell at compile time whether or not the index will be within bounds.

In case a compile time constant expression evaluates to `true`, the static assertion should not abort the compilation process and in the optimal case should also not create any additional overhead by adding code that has to be evaluated at runtime. In case an expression evaluates to `false`, the static assertion should abort the compilation process immediately with an error message that should indicate as accurately as possible what went wrong.

Unfortunately, static asserts are not part of the C++ programming language (otherwise this would be something to be explained in every C++ primer). As a result, it is up to the programmer to simulate static assertions as good as possible using the language features available. Obviously it is necessary to use the result of the compile time constant expression to generate either perfectly valid code in case the expression evaluates to `true`, or to generate invalid code that the compiler rejects by emitting an error message.

There are several ways to create invalid code based on the result of an expression [2]. Several of these ideas work for both C and C++, whereas one of these ideas is restricted to C++ only. The most commonly used approach is to declare an array of either size 1 in case the expression is `true`, or 0 in case the expression evaluates to `false`:

Listing 4: First approach for static assertions

```
1 #define STATIC_ASSERT( expression ) \
2    int array[(expression)?(1):(0)]
3
4 // ...
5
6 STATIC_ASSERT( sizeof(int) < sizeof(short) );  // Compile time error
```

[2] The approaches illustrated in this report are taken from [Wil05].

Depending on the result of the expression, the size of the array will be either 1 or 0. Since an array size of zero is invalid in both C and C++, the compiler will issue an error in case the result of the given expression is `false`. Note that the definition of the static assert in this way will limit its usability since it introduces the variable `array` into the current scope and also potentially adds code overhead. Additionally, although arrays of size 0 are invalid some compiler merely issue a warning instead of a compilation error. The following scheme resolves both of these issues:

Listing 5: Improvement of the first approach for static assertions

```
#define STATIC_ASSERT( expression ) \
   do { typedef int array[(expression)?(1):(-1)]; } while(0)
```

The array is wrapped in the scope of a do-while-loop, that is at maximum executed once. However, due to the single type definition inside the loop, the compiler can easily remove it from the executable. Additionally, instead of creating an array of size 0 in case of a `false` expression, an array of size -1 is created, which causes all compilers to abort the compilation process.

A second approach to generate invalid code in case of a `false` expression is based on switch statements. In this case, the generation of invalid code is based on the requirement that each case clause must have a different value [Wil05]:

Listing 6: Second approach for static assertions

```
#define STATIC_ASSERT( expression ) \
   switch( 0 ) { case 0: case (expression):; }
```

A third approach relies on the fact that bitfields must have a size of at least one bit [Wil05]:

Listing 7: Third approach for static assertions

```
#define STATIC_ASSERT( expression ) \
   struct x { unsigned int v : (expression); }
```

Another idea is based on the compile time instantiation of a template specialization. Therefore this approach is only suited for C++ codes:

Listing 8: Fourth approach for static assertions

```
template< bool > struct STATIC_ASSERT_FAILED;

template<> struct STATIC_ASSERT_FAILED<true> {
   enum { value = 1 };
};

#define STATIC_ASSERT( expression ) \
   { STATIC_ASSERT_FAILED<expression> assert_failed; }
```

The idea here is to first declare a template class with a single boolean template argument and to specialize this class for a value of `true`. In case the expression evaluates to `true`, the specialization is selected and the compiler creates an object of type `STATIC_ASSERT_FAILED<true>` within a new scope. In case the expression evaluates to `false`, the compiler tries to instantiate `STATIC_ASSERT_FAILED` for `false`, but only has the declaration available. Therefore the compilation process is aborted with the error message that no definition for `STATIC_ASSERT_FAILED<false>` can be found.

The static assertion in the Boost library is based on the last version of `STATIC_ASSERT`. However, the Boost library improves the idea in one very important aspect: most compilers will complain about the unused and unreferenced object `assert_failed` [3]. Therefore the Boost library transforms the definition of a temporary object into a type definition:

Listing 9: The Boost static assertion

```
namespace boost {

#define BOOST_JOIN( X, Y ) BOOST_DO_JOIN( X, Y )
#define BOOST_DO_JOIN( X, Y ) BOOST_DO_JOIN2(X,Y)
#define BOOST_DO_JOIN2( X, Y ) X##Y

```

[3] At least if all warnings are enabled.

```
template< int > struct BOOST_STATIC_ASSERT_TEST {};

template< bool > struct STATIC_ASSERT_FAILURE;
template<> struct STATIC_ASSERT_FAILURE<true> { enum { value = 1 }; };

#define BOOST_STATIC_ASSERT( expression ) \
   typedef \
      BOOST_STATIC_ASSERT_TEST< \
         sizeof( STATIC_ASSERT_FAILURE< (bool)(expression) > ) > \
      BOOST_JOIN( BOOST_STATIC_ASSERT_TYPEDEF, __LINE__ )

} // namespace boost
```

The basic idea of selecting either the defined template specialization for `true` expressions and to abort the compilation in case the undefined template is required for `false` expressions is still the same as before. However, since instead of an object in this approach a type definition is created, the definition of this type alone would not suffice:

Listing 10: Instantiation of the STATIC_ASSERT_FAILURE template

```
typedef STATIC_ASSERT_FAILURE<true>  BOOST_STATIC_ASSERT_TYPEDEF_1;  // No compiler error
typedef STATIC_ASSERT_FAILURE<false> BOOST_STATIC_ASSERT_TYPEDEF_2;  // No compiler error!
```

For the type definition alone, the template doesn't have to be instantiated. The compiler accepts both of these type definitions, even though the second type definition is based on the undefined base template. Therefore even for a `false` expression this would not result in a compilation error. In order to force the instantiation of the `STATIC_ASSERT_FAILURE` template, two ideas can be used: either the `value` member enumeration of the `STATIC_ASSERT_FAILURE` template is accessed and used to instantiate the `BOOST_STATIC_ASSERT_TEST` template. Now the compiler has to instantiate the template in order to check the compile time constant value of the member enumeration to be able to create the according type definition. The second idea is to evaluate the size of the `STATIC_ASSERT_FAILURE` type. This also forces the compiler to instantiate the type. However, in order to be able to define a new type in a type definition, the `BOOST_STATIC_ASSERT_TEST` still has to be wrapped around the construct. Note that the argument to the `BOOST_STATIC_ASSERT` is explicitly cast to `bool` using old-style casts. The reason for this is that currently too many compilers have problems with `static_cast` when used inside integral constant expressions.

The second part of the `BOOST_STATIC_ASSERT` type definition is the `BOOST_JOIN` macro. This piece of macro magic joins the two arguments together, even when one of the arguments is itself a macro (see section 16.3.1 of the C++ standard [Str03]). The key is that macro expansion of macro arguments does not occur in `BOOST_DO_JOIN2` but does in `BOOST_DO_JOIN` [Boo].

3 Comprehensible Error Messages

Although these techniques allow for the detection of compile time errors and the abortion of the compilation process, one major problem remains: comprehensible error messages. Consider the static assertion as introduced in Listing 5 by creating an array of negative length in case the compile time static expression evaluates to `false` in the following example:

Listing 11: First example for bad error messages

```
#define STATIC_ASSERT( expression ) \
   do { int array[(expression)?(1):(-1)]; } while(0)

int main() {
   // ...
   STATIC_ASSERT( sizeof(double) < 1 );
   // ...
}
```

The example is constructed such that the expression will always evaluate to `false`. What we would hope for is a compilation error telling us that the size of a double value will never be smaller than one byte. However, imagine the poor programmer confronted with the following error message (G++ 4.2.1):

```
Error: Size of array "array" is negative
```

Here the compiler is perfectly right in complaining about an invalid, negative array size. However, the error message is in no way corresponding to the actual error, i.e., that our assumption that a `double` value has less than one byte will always be wrong.

Note that depending on the compiler, we can improve the situation slightly. If the compiler (as in the case of the GCC compiler) includes the name of the array in the error message, we can use this to give the programmer the information that this particular error corresponds to a failed compile time assertion:

Listing 12: Second example for bad error messages

```
1 #define STATIC_ASSERT( expression ) \
2    do { int STATIC_ASSERTION_FAILED[(expression)?(1):(-1)]; } while(0)
```

Now the compiler will give us the following error message:

```
Error: Size of array "STATIC_ASSERTION_FAILED" is negative
```

This improves the situation somewhat: although this error message still does not contain any information about the actual error, it points to the fact that a static assertion has failed.

Using the `BOOST_STATIC_ASSERT`, the error messages read for instance as following:

```
Error: Invalid use of "sizeof" for incomplete type
         "boost::STATIC_ASSERTION_FAILURE<false>"
```

In this case, the error message contains the name of the template instance the compiler failed to instantiate. Again, the name is not corresponding to the actual error, but a programmer gets the idea that a compile time assertion failed due to an invalid condition. Although the kind of the error is still a mystery, the programmer gets a hint of what to expect if he looks for the source of the error.

As already stated, compile time constraints are not part of the C or C++ programming languages. Therefore, although these error messages are far from being precisely aiming at the actual error, this is the best that we can get by using the available language features. An important realization is that we have to name our variables/class accordingly and have to hope that the compiler will print the name in the corresponding error message to give us a hint of the source of the error.

4 The *pe* Compile Time Constraints

The *pe* compile time constraints are an extension of the ideas of `BOOST_STATIC_ASSERT`. As illustrated in the last section, the only drawback of this macro is the comprehensibility of the resulting error messages that always point out the instantiation failure of the undefined `STATIC_ASSERT_FAILURE` base template. Therefore the consequent continuation of this scheme is to create several specific `FAILURE` classes that point to an individual error as closely as possible. The following example demonstrates this for the `pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE` compile time constraint:

Listing 13: Example for a *pe* compile time constraint

```
1  // This class is used as a wrapper for the instantiation of the specific constraint
2  // class templates. It serves the purpose to force the instantiation of either the
3  // defined specialization or the undefined basic template during the compilation. In
4  // case the compile time condition is met, the type pe::CONSTRAINT_TEST<1> is defined.
5  template< int > struct CONSTRAINT_TEST {};
6
7  // Helper template class for the compile time constraint enforcement. Based on the
8  // compile time constant expression used for the template instantiation, either the
9  // undefined basic template or the specialization is selected. If the undefined basic
10 // template is selected, a compilation error is created.
11 template< bool > struct CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED;
12 template<> struct CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED<true> {
13    enum { value = 1 };
14 };
15
16 // In case the given data type \a T is not a floating point data type, a compilation
17 // error is created.
18 #define pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE(T) \
19    typedef \
20       pe::CONSTRAINT_TEST< \
```

```
21          pe::CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED< \
22             boost::is_floating_point<T>::value >::value > \
23       BOOST_JOIN( CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_TYPEDEF, __LINE__ )
```

All *pe* constraints work according to this pattern: the `pe::CONSTRAINT_TEST` template is used as a wrapper around the instantiation of the accordingly named constraint class. These classes are always named in capital letters in order to stand out in the resulting error messages and such that the error is described as accurately as possible. Additionally, as a convention, all compile time constraints use the prefix `CONSTRAINT_` to indicate the purpose of these classes.

In contrast to the Boost static assertion, the *pe* employs the strategy to use the member enumeration `value` of the corresponding constraint class to instantiate the `CONSTRAINT_TEST` and does not use the `sizeof` operator. Also in contrast to Boost, the compile time test is already incorporated in the constraint. In this example, we can use one of the Boost type traits, `boost::is_floating_point` [Boo]. In case the given data type is a (possibly cv-qualified) floating point data type, the `value` member enumeration of the type trait evaluates to `true`, otherwise to `false`.

The instantiation of the constraint class is wrapped in a type definition in an accordingly named macro. By convention, the macro is named after the constraint class and has an additional prefix `pe_` to indicate its origin and to minimize name clashes.

This compile time constraint is for instance used in the `Quaternion` class of the math library of the *pe*:

Listing 14: Application of a *pe* compile time constraint

```
1 template< typename Type >
2 class Quaternion {
3    // ...
4    Type values_[4];
5    // ...
6    pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE( Type );
7    // ...
8 };
```

The `Quaternion` class is used for representing the orientation of rigid bodies in a rigid multibody simulation. The determinant of the `Quaternion` will always be 1. Therefore the four values of the `Quaternion` have to be in the range [0..1] during the entire simulation and independent of the current orientation of the objects. In consequence, the values of a `Quaternion` can only be floating point values. This design is enforced by the `pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE` compile time constraint. Due to the compile time constraint, it is not possible to instantiate the `Quaternion` with any data type except built-in (fundamental) floating point data types (i.e., `float`, `double`, and `long double`).

By the time of this writing, the *pe* altogether features 44 compile time constraints. The following list gives a complete overview of the available constraints. As you will note, in most cases every constraint is available in a positive and a negative formulation, as for instance the `pe_CONSTRAINT_MUST_BE_POD` and `pe_CONSTRAINT_MUST_NOT_BE_POD`. This results in several constraints with similar meanings, but allows for a very flexible selection of constraints according to the current programming context. For instance, `CONSTRAINT_MUST_BE_BASE_OF` and `CONSTRAINT_MUST_BE_DERIVED_FROM` can be interchanged by switching the arguments, but depending on the context the constraint is used one or the other may be preferable.

- `pe_CONSTRAINT_MUST_BE_ARITHMETIC_TYPE(T)`
- `pe_CONSTRAINT_MUST_NOT_BE_ARITHMETIC_TYPE(T)`
- `pe_CONSTRAINT_MUST_BE_BASE_OF(B, D)`
- `pe_CONSTRAINT_MUST_NOT_BE_BASE_OF(B, D)`
- `pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF(B, D)`
- `pe_CONSTRAINT_MUST_NOT_BE_STRICTLY_BASE_OF(B, D)`
- `pe_CONSTRAINT_MUST_BE_BOOLEAN_TYPE(T)`
- `pe_CONSTRAINT_MUST_NOT_BE_BOOLEAN_TYPE(T)`
- `pe_CONSTRAINT_MUST_BE_BUILTIN_TYPE(T)`
- `pe_CONSTRAINT_MUST_NOT_BE_BUILTIN_TYPE(T)`
- `pe_CONSTRAINT_MUST_BE_CLASS_TYPE(T)`

- pe_CONSTRAINT_MUST_NOT_BE_CLASS_TYPE(T)
- pe_CONSTRAINT_POINTER_MUST_BE_COMPARABLE(P1, P2)
- pe_CONSTRAINT_MUST_BE_CONST(T)
- pe_CONSTRAINT_MUST_NOT_BE_CONST(T)
- pe_CONSTRAINT_POINTER_MUST_BE_CONVERTIBLE(FROM, TO)
- pe_CONSTRAINT_MUST_BE_DERIVED_FROM(D, B)
- pe_CONSTRAINT_MUST_NOT_BE_DERIVED_FROM(D, B)
- pe_CONSTRAINT_MUST_BE_STRICTLY_DERIVED_FROM(D, B)
- pe_CONSTRAINT_MUST_NOT_BE_STRICTLY_DERIVED_FROM(D, B)
- pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE(T)
- pe_CONSTRAINT_MUST_NOT_BE_FLOATING_POINT_TYPE(T)
- pe_CONSTRAINT_MUST_BE_INTEGRAL_TYPE(T)
- pe_CONSTRAINT_MUST_NOT_BE_INTEGRAL_TYPE(T)
- pe_CONSTRAINT_MUST_BE_NUMERIC_TYPE(T)
- pe_CONSTRAINT_MUST_NOT_BE_NUMERIC_TYPE(T)
- pe_CONSTRAINT_MUST_BE_POD(T)
- pe_CONSTRAINT_MUST_NOT_BE_POD(T)
- pe_CONSTRAINT_MUST_BE_POINTER_TYPE(T)
- pe_CONSTRAINT_MUST_NOT_BE_POINTER_TYPE(T)
- pe_CONSTRAINT_MUST_HAVE_SAME_SIZE(T1, T2)
- pe_CONSTRAINT_MUST_NOT_HAVE_SAME_SIZE(T1, T2)
- pe_CONSTRAINT_MUST_BE_SAME_TYPE(A, B)
- pe_CONSTRAINT_MUST_NOT_BE_SAME_TYPE(A, B)
- pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE(A, B)
- pe_CONSTRAINT_MUST_NOT_BE_STRICTLY_SAME_TYPE(A, B)
- pe_CONSTRAINT_MUST_HAVE_SIZE(T, S)
- pe_CONSTRAINT_MUST_NOT_HAVE_SIZE(T, S)
- pe_CONSTRAINT_MUST_BE_SUBSCRIPTABLE(T)
- pe_CONSTRAINT_MUST_BE_SUBSCRIPTABLE_AS_DECAYABLE_POINTER(T)
- pe_CONSTRAINT_SOFT_TYPE_RESTRICTION(T, TYPELIST)
- pe_CONSTRAINT_TYPE_RESTRICTION(T, TYPELIST)
- pe_CONSTRAINT_MUST_BE_VOLATILE(T)
- pe_CONSTRAINT_MUST_NOT_BE_VOLATILE(T)

5 The MUST_BE_BASE_OF constraints

Let us highlight some of the more interesting constraints. The `CONSTRAINT_MUST_BE_BASE_OF` constraint enforces that a certain class must be a public or non-public direct or indirect base class of a second class:

Listing 15: The MUST_BE_BASE_OF constraint

```
class A {};
class B : public A {};
class C : public B {};
class D : private A {};
class E;

int main()
{
    pe_CONSTRAINT_MUST_BE_BASE_OF( A, A );  // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF( A, B );  // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF( A, C );  // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF( A, D );  // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF( A, E );  // Compilation error!
}
```

Only the fifth constraint will abort the compilation to report that `E` is not related to `A`. Note that this constraint allows for the test whether `A` is a base of `A`, i.e., `A` can be seen as directly related to `A`. In contrast to this constraint, the `CONSTRAINT_MUST_BE_STRICTLY_BASE` constraint creates a compilation error in case the two given types are equal:

Listing 16: The MUST_BE_STRICTLY_BASE_OF constraint

```
class A {};
class B : public A {};
class C : public B {};
class D : private A {};
class E;

int main()
{
   pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF( A, A );  // Compilation error!
   pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF( A, B );  // No compilation error
   pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF( A, C );  // No compilation error
   pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF( A, D );  // No compilation error
   pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF( A, E );  // Compilation error!
}
```

Internally this is solved by the concatenation of several Boost type traits:

Listing 17: Implementation of the MUST_BE_BASE constraints

```
#define pe_CONSTRAINT_MUST_BE_BASE_OF(B,D) \
  typedef \
    CONSTRAINT_TEST< \
      CONSTRAINT_MUST_BE_BASE_OF_FAILED< is_base_of<B,D>::value >::value > \
    BOOST_JOIN( CONSTRAINT_MUST_BE_BASE_OF_TYPEDEF, __LINE__ )

#define pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF(B,D) \
  typedef \
    CONSTRAINT_TEST< \
      CONSTRAINT_MUST_BE_STRICTLY_BASE_OF_FAILED< is_base_of<B,D>::value && \
                                                  !is_base_of<D,B>::value >::value > \
    BOOST_JOIN( CONSTRAINT_MUST_BE_STRICTLY_BASE_OF_TYPEDEF, __LINE__ )
```

6 The MUST_BE_SAME_TYPE constraints

Two other interesting constraints are focused on evaluating at compile time if two given data types are the same: `pe_CONSTRAINT_MUST_BE_SAME_TYPE` and `pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE`. The difference between these two constraints is that the former ignores any cv-qualifiers on both types, whereas the latter one takes them into account:

Listing 18: The MUST_BE_SAME_TYPE constraints

```
int main()
{
   pe_CONSTRAINT_MUST_BE_SAME_TYPE( int, int );           // No compilation error
   pe_CONSTRAINT_MUST_BE_SAME_TYPE( const int, int );     // No compilation error
   pe_CONSTRAINT_MUST_BE_SAME_TYPE( int, volatile int );  // No compilation error
   pe_CONSTRAINT_MUST_BE_SAME_TYPE( float, double );      // Compilation error!

   pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( int, int );           // No compilation error
   pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( const int, int );     // Compilation error!
   pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( int, volatile int );  // Compilation error!
   pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( float, double );      // Compilation error!
}
```

In order to test whether two types are the same, the Boost library offers the `is_same` type trait. However, this type trait always takes the cv-qualifiers into account and would therefore only be suited for the second constraint. Therefore, the *pe* implements two additional type traits according to the naming convention of the *pe*:

Listing 19: The IsSame type traits

```
template< typename A, typename B >
struct IsSame
{
 private:
   class No {};
   class Yes { No no[2]; };

   static A&  createA();
```

```
9     static B&  createB();
10
11    static Yes testA( const volatile B& );
12    static No  testA( ... );
13    static Yes testB( const volatile A& );
14    static No  testB( ... );
15
16  public:
17    enum { value = ( sizeof( testA( createA() ) ) == sizeof( Yes ) ) &&
18                   ( sizeof( testB( createB() ) ) == sizeof( Yes ) ) };
19 };
20
21 template< typename A, typename B >
22 struct IsStrictlySame
23 {
24    enum { value = 0 };
25 };
26
27 template< typename T >
28 struct IsStrictlySame<T,T>
29 {
30    enum { value = 1 };
31 };
```

The `IsSame` type trait is defined according to ideas of Andrei Alexandrescu [Ale01] and Herb Sutter [Sut07] to test the relationship of two data types. The `IsStrictlySame` type trait relies on the strict type matching of template instantiations to test whether two types are exactly the same.

Using these two type traits, the two constraints can be implemented as follows:

Listing 20: Implementation of the MUST_BE_SAME_SIZE constraints

```
1  #define pe_CONSTRAINT_MUST_BE_SAME_TYPE(A,B) \
2    typedef \
3      CONSTRAINT_TEST< \
4        CONSTRAINT_MUST_BE_SAME_TYPE_FAILED< IsSame<A,B>::value >::value > \
5      BOOST_JOIN( CONSTRAINT_MUST_BE_SAME_TYPE_TYPEDEF, __LINE__ )
6
7  #define pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE(A,B) \
8    typedef \
9      CONSTRAINT_TEST< \
10       CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE_FAILED<IsStrictlySame<A,B>::value >::value > \
11     BOOST_JOIN( CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE_TYPEDEF, __LINE__ )
```

7 Conclusion

In this report we have introduced the *pe* compile time constraint library. These compile time constraints combine the ability to detect compile time errors and in consequence abort the compilation process and comprehensible error messages. These compile time constraints give a programmer a powerful tool to check for compile time errors, or to enforce certain design decisions.

References

[Ale01] A. Alexandrescu, *Modern C++ Design, Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

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

[IR09] K. Iglberger and U. Rüde, *The pe Rigid Multi-Body Physics Engine*, Tech. report, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2009.

[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.

[Wil05] M. Wilson, *Imperfect C++*, Addison-Wesley, 2005.