# SPECIFYING SELECTION CRITERIA USING C++ EXPRESSION TEMPLATES

C. D. Jones*, Cornell University, Ithaca, NY 14853 USA

*Abstract*

Generic programming, as exemplified by the C++ standard library, makes use of functions or function objects (objects that accept function syntax) to specialize generic algorithms for particular uses. Such separation improves code reuse without sacrificing efficiency. We employed this technique in our combinatoric engine, DChain, in which physicists combine lists of child particles to form a list of parent hypotheses, e.g.,

d0 = pi.plus() * K.minus();

The selection criteria for the hypothesis is defined in a function or function object that is passed to the list's constructor.

However, C++ requires that functions and class declarations be defined outside the scope of a function. Therefore physicists are forced to separate the code that defines the combinatorics from the code that sets the selection criteria. We will discuss a technique using C++ expression templates to allow users to define function objects using a mathematical expression directly in their main function, e.g.,

func = sqrt( beamEnergy*beamEnergy - vPMag*vPMag ) >= 5.1*k_GeV.

Use of such techniques can greatly decrease the coding 'excise' needed to perform an analysis.

## INTRODUCTION

One of the hoped for benefits of the transition from FORTRAN to C++ was the ability to create libraries allowing physicists to write programs in a more intuitive and compact form. One such example is the ability to add four vectors using the '+' operator. However, C++ allows a library developer to arbitrarily assign meaning to all the definable operators, allowing one to create a whole new compact syntax for specialized purposes. One area where we have applied this technique is our combinatoric engine, DChain [1].

In DChain, physicists combine lists of child particles to form a list of parent hypotheses, e.g.,

D0_list = pi_list.plus() * K_list.minus();

The selection criteria for the hypothesis is defined in a function or function object (an object that accepts function syntax) that is passed to the list's constructor. This follows the idiom used by the C++ standard library, where generic algorithms are specialized by passing them a function or function object.

However, C++ requires that functions and class declarations be outside the scope of any function. Therefore, physicists are forced to separate the code that defines the combinatorics from the code that sets the selection criteria. We were able to alleviate this restriction in most cases by using the C++ expression template technique to allow users to define unnamed function objects using a mathematical expression directly in their main function. A working snippet of code is shown below:

```
Var< mass   >            vMass;
Var< energy >            vEnergy;
Var< p_mag >             vPMag;
SimpleSelector<Decay> select_D0 =
  abs(vMass – kD0Mass)          < 100*k_MeV &&
  abs(vEnergy – beamEnergy)     < 100*k_MeV &&
  abs( sqrt( beamEnergy*beamEnergy –
     vPMag*vPMag) – kD0Mass) < 10*k_MeV);

DecayList D0_list( select_D0 );
D0_list = pi_list.plus() * K_list.minus();
```

In this example we are trying to find $D^0$s that decay to $p^+$ $K^-$ (and the charge conjugate). We require that the $D^0$ candidates have a mass within 100 MeV of the nominal mass, an energy within 100 MeV of the accelerator's beam energy, and have a 'beam constrained mass' (mass calculated using the beam energy instead of the measured object energy) within 10 MeV of the nominal mass. This is an extremely compact (and hopefully readable) way of expressing such an operation.

## EXPRESSION TEMPLATES

Expression templates use two of C++'s features: operator overloading and templates. The idea is to convert an expression (usually expressed in mathematical notation) into a new class type that encapsulates the expression as a graph.

Consider the expression

sqrt( a*a – b*b ) >= v.

This can be expressed as the graph seen in Figure 1.

At the bottom of the graph are the variables **a**, **b** and **v**. In the next level up **a** and **b** are combined with themselves using the binary multiplication operation. The multiplications are both done first since multiplication has a higher precedence rank than subtraction. In the next level, the results of the multiplications are passed to the binary subtraction operation. In the penultimate level, the result of the subtraction is passed to the unary square root, **sqrt**, method. In the ultimate level, the result of the square root method, as well as the remaining variable **v,**

are passed to the binary greater-than-or-equal-to operator (**>=**).

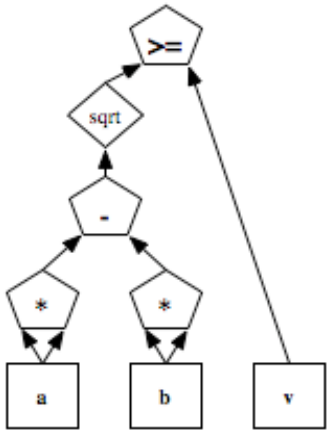


Figure 1: Mathematical expression as a graph

In a standard C++ implementation, the operators *****, **-**, >= and the method **sqrt** would be overloaded to take as arguments the type of the class of the variables **a**, **b**, and **v**. The operators *****, **-** and the method **sqrt** would also return a new instance of that same class (>= would return a bool). However, passing temporary instances of a class as intermediate results of the full expression can be costly (e.g., in the case of a 100 item list). Expression templates alleviate that problem.

If we were to apply expression templates to our example, the operators ***** and – and the method **sqrt** would not return an instance of the same class as our input variables. Instead, they would return a new templated class where the class represents the specific operation to be performed while the template arguments denote upon what the operation should be performed. This also means that all the operators and methods used in the expression are templated based on all of their arguments. In that case, the declaration of our functions would look like

- template<typename T, typename S>
  **MultOp**<T,S>
  **operator***(const T&, const S&);
- template<typename T, typename S>
  **SubOp**<T,S>
  **operator-**(const A&, const B&);
- template<typename T>
  **SqrtOp**<T>
  **sqrt**(const T&);
- template<typename T, typename S>
  **GtEqOp**<T,S>
  **operator>=**(const T&, const S&);

The returned objects would hold copies of (or references to) the variables actually passed to the functions.

Assuming that our variables are of type **A**, **B** and **V**, the type of the class returned by our expression would be

```
GtEqOp<
        SqrtOp< SubOp<
                        MultOp<A,A>,
                        MultOp<B,B> > >,
        V>
```

To get the value of the expression, you would call a method of this class (say **operator ()** ). This method would call the same named method on its internal copies of the intermediate variables and would take the results of those calls and apply its own operation on them, returning the result of that operation. For example, calling operator() on our **GetEqOp<…>** object would cause **SqrtOp<…>::operator()** to be called (which would further descend down the call graph) as well as **v.operator(),** and then those two values would be compared using the standard >= operation.

So how is this an improvement? Let us take the specific example that our variables are instances of our own container class **Vector** (similar to a **std::vector**), with all containers having the same length, and our functions are meant to be applied to each element of the list. For the standard implementation it would be necessary to have five loops over the length of the containers. With expression templates, if the Operation class's **operator()** takes a container index as an argument, then the entire expression could be done with just one loop. In fact, if the functions were declared inline, then an optimizing compiler could reduce the filling of a new **Vector** which is defined as

```
class Vector {
…
 template <typename Node>
 void operator=(const Node& iN) {
    for(int i=0; i<size; ++i){ *this(i) = iN(i);} }
```

to machine code equivalent to

```
    for(int i=0; i < size; ++i) {
        *this(i)= sqrt( a(i)*a(i) - b(i)*b(i) ) >= v(i);}
```

## *Technical Caveats*

Use of overloaded template operators in the global namespace is very dangerous. The problem is the C++ rules for choosing an overloaded function in which templates have a higher priority than argument coercion (i.e., automatically converting an argument of one type into a compatible type such as through a non-explicit constructor that takes one argument). So in our above example, if **operator*** was declared in the global namespace then that function would be used if you attempted to multiply an int by a float instead of having the compiler convert the int to a float and then multiply the two floats. This problem can be avoided by defining the operators and functions to be overloaded in the same namespace as the classes to be used as variables in the expression, in which case C++ can use the Koenig lookup rule [2]. This rule states that when searching for overloaded operators or functions, C++ will first look in

the namespace of the arguments to the operator or function, even if the function has not been explicitly stated as coming from that namespace (operators have no way of specifying an explicit namespace).

Sometimes it is not possible for the class used for the 'variables' of the expression to have the same interface as the Operation classes. In that case, a wrapper Operation class could be used to adapt the variable class to the necessary interface. To make use of the wrapper transparent to the user we would need to hide the conversion by using one of two types of specialization.

The simplest is to use function specialization. For each possible argument permutation involving one of the 'variable' classes, we write a special function to explicitly create the wrapper class and pass it and any other function arguments to the non-specialized form of the function. This works for almost all modern compilers, but the amount of code one must write grows factorially with the number of arguments. For instance, for a method taking two arguments one must write three specialized forms (one with only the first argument specialized, one with only the second argument specialized and one with both specialized).

The more indirect way is to use specialization techniques from template meta-programming. The idea is to use a new intermediate class to determine exactly what class type to use for the return value related to one particular argument, instead of using the argument itself. This is done by creating the class

```
template< typename T> struct MakeWrapper
    { typedef T Return; };
template<> struct MakeWrapper<Var>
    { typedef Wrapper< Var > Return; };
```

Then, when defining the function's return value one uses **MakeWrapper<T>::Return** in place of the actual type T of the argument passed to the function. The advantage to this approach is you only have to write one version of the function, and if you add a new argument type all you need to do is create a new **MakeWrapper** specialization and all of your functions will handle the new case correctly. The disadvantage is that this relies on the compiler to properly handle a complex template instantiation, which not all modern compilers do.

## IMPLEMENTATION

The design of the selection functional classes was inspired by the wonderful lambda package, which is part of the boost library [3]. The idea is to use objects to represent the variables of the expression. When the expression is actually evaluated, these 'variable' objects return the value they represent. In the case of the lambda package, the 'variables' are placeholders for the argument list of the function being defined (i.e., they represent the first, second, or third variable in the argument list). In the case of DChain's selection functions there is only one argument (the item in the list to be evaluated), so the 'variable' placeholders represent functions to call on that argument in order to get the proper value. For example, if the selection function operates on **Decay**s, one might have one 'variable' object that returns the **Decay**'s mass and another object that returns the **Decay**'s momentum.

The 'variable' object is the templated class **DChain::Var**. The template argument must be a class that has an **operator()** taking the appropriate type and returning a double (since for now we evaluate the expression to a double). So if our **Decay** class inherits from the class **Candidate** which has a method **mass**, we could use the standard C++ library functional adapter classes to write a 'variable' object that calls **mass** as

```
DChain::Var<
    const_mem_fun_ref_t<double,Candidate> >
vMass( mem_fun_ref(&Candidate::mass ) );
```

Here we say that **vMass** is a **DChain::Var<>** using a functional object to hold a const member function of the class **Candidate,** where the member function returns a double. The constructor for **vMass** takes an instance of the functional object that has been told to call the **Candidate::mass** method. Such a specification is very cumbersome to type. So instead, a small specialized **DChain::mass** class was created that explicitly calls **Candidate::mass**, allowing one to write

```
DChain::Var< DChain::mass > vMass;
```

**DChain::Var** has no methods, it just has a public variable holding the function to call. **DChain::Var**'s sole purpose is to make sure the correct overloaded functions are used when the compiler evaluates the expression.

### *Mathematical Operators*

The implementation of the overloaded math operations all follow a common pattern. There are three overloaded versions: two that have one **Var<>** argument and one double argument, with the third taking two **Var<>** arguments. One of these functions for addition is below.

```
template<class F>
Var<Composite<F, binder2nd<plus<double>>>>
  operator+(const Var<F >& iVar, double iValue )
 { typedef Composite<F, binder2nd<plus<double>>>
            CompT;
   CompT temp( iVar.m_func,
                bind2nd(plus<double>(), iValue ) );
   return Var<CompT >( temp );  }
```

I make heavy use of standard C++ adapters in the definition of **operator+**. The return value of **operator+** is a **Var<>** holding a **Composite<>** object which passes the results of **F::operator()** into the **operator()** method of the **binder2nd<plus<double> >** class. The **binder2nd** class holds the value **iValue** and passes that in as the second argument to **plus<double>::operator()** while passing the value given to its own **operator()** as the first

argument. This all boils down to creating a functional object that multiplies the result of **iVar.operator()** with **iValue**. Specialized versions of the functions **abs** and **sqrt** are defined in a similar manner.

### *Comparison Operators*

The ultimate goal is to create a selection functional object that evaluates to true or false. To that end, the comparison operators (<, <=, >, >=, ==, !=) are templated functions taking two arguments, one a **Var<>** and the other a **double** or **bool** (for == and !=). The operators return an instance of the templated class **VarComparisonMethod<>** (**VCM<>**), where the template argument is the argument used for the **Var<>**. The **VCM<>** holds the function to apply (from the **Var<>**), the comparison operation to perform, and the **double** or **bool** value to compare against.

### *Compound Comparisons*

In the majority of cases, selecting an item (such as a Track) requires several different comparisons to be made (e.g., minimum values for $X^2$ of fit and magnitude of the momentum). To that end, the **&&** and **||** operators are overloaded to take **VCM<>**s as arguments and return a **MethodOr<T,S>** or **MethodAnd<T,S>**. The two template arguments **T** and **S** are just the arguments passed to the **&&** and **||** operators. This allows the following compound statement

```
abs(vMass-5.28)<.1 && vPMag > 1.0
```

### *Expressions and Selection Objects*

Some selections require more expressive power than a mathematical expression can give (e.g., looping may be needed). Those cases are best handled by special purpose selection objects. However, the less a class does the easier it is to debug and to reuse. Therefore the **&&** and **||** operators also allow selection objects as arguments, e.g.,

```
MySelector mySelector;
SimpleSelector<Track> pionSel = mySelector &&
                                vPMag > 0.2;
```

This works by having operator **&&** and || automatically create wrappers for any object that inherits from **DCSelectionFunction<T>**. This is done using the **MakeWrapper** technique discussed earlier. Detecting inheritance is done using the technique discussed in [4].

### *Adapting to DChain*

DChain expects its selection object to inherit from the abstract class **DCSelectionFunction<T>**. Instances of selection objects are passed to the list and the list holds a pointer to that object. Therefore the selection objects must live at least until they are used by the list to make its selection. Temporaries on the stack have sufficient lifetimes for this purpose. However, the type returned by the comparison expression is extremely complex (because the expression graph is contained in the template arguments) and therefore should not be seen by users. So what type should we use for our temporary that 'pins' the expression object in memory? There are two choices: a reference or a wrapper.

If the type returned by the comparison expression inherited from **DCSelectionFunction<T>,** then we could use a reference for our temporary, e.g.,

```
DCSelectionFunction<T>& d0Sel = …;
```

Unfortunately, this syntax can be confusing to users (i.e., "what is that '&' and when should I use it"?) and the compiler error message generated when the '&' is missing is not very informative.

The alternative chosen was to use an adapter class **SimpleSelector<T>** that inherits from **DCSelectionFunction<T>**. In addition, all types returned from a comparison expression inherit from the **MethodBase<T>** class. The constructor for **SimpleSelector<T>** takes a **MethodBase<T>** as an argument and asks **MethodBase** to clone itself, so a copy of that object is placed on the heap. The **SimpleSelector<T>** then manages the lifetime of that clone. Although this method makes use of the (potentially expensive) operator **new**, we have found that our time is still dominated by the actual combinatorics.

## CONCLUSION

C++ expression templates are an extremely powerful tool for creating libraries allowing very expressive syntax while not precluding optimal runtime performance. Use of this technique is technically challenging, but I believe it is well worth the effort if it allows physicists to succinctly express their selections, thereby decreasing the time it takes to write their code, while making that code more understandable.

## ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation.

## REFERENCES

[1] S. Patton and C.D. Jones. DCHAIN- combinatorics and conjugation made easy. In *International Conference on Computing in High-Energy Physics and Nuclear Physics (CHEP 1998), Chicago, Il,* August 1998.

[2] C++ Standard ISO/IEC 14882 1998(E) section 3.4.2

[3] http://www.boost.org

[4] A. Alexandrescu, "Modern C++ Design" *Addison-Wesley*, 2001