GCombiner 1.0

HERA-B Note 01-001

Thorsten Glebe
Max-Planck-Institut für Kernphysik, Heidelberg
T.Glebe@mpi-hd.mpg.de

January 18, 2002

Abstract

GCombiner is a universal concept for combining arbitrary objects of same type from an arbitrary amount of containers. The usage of combiner objects is similar to STL containers. GCombiner's are independend from ROOT and ARTE and can be used in any environment.

Contents

1 Introduction **1**

2 Code and compilation **2**

3 A simple example **2**

4 Using GCombiner **4**
- 4.1 Adding more containers 5
- 4.2 Accessing combination elements 5
- 4.3 Selecting combinations 7
- 4.4 Configure GCombiners 9

5 Using SCombiner **9**
- 5.1 Generate a Combiner object 9
- 5.2 Accessing objects in a combination 10
- 5.3 Accessing the complementary objects to a combination . . . 11
- 5.4 Configure SCombiner 12

1 Introduction

In data analysis it is a common task to form combinations of objects in order to reconstruct properties of the event. A common example is the search for secondary vertices where track tuples are formed in order to compute a vertex. As it is not clear from the beginning which tracks belong to a vertex, an iterative procedure must try combinations of tracks in order to find a proper vertex.

The problem of combining objects appears in many different situations, however from the programmer's point of view the code is almost more or less the same. Therefore a set of template classes have been developed which allow to combine objects of arbitrary (but same) type from an arbitrary amount of lists. The behaviour of the GCombiner classes can be adopted for different object types and the user can add his own object collision detection to throw out unwanted combinations.

A GCombiner is simple to use and resembles a STL [1] container. The GCombiner classes are CINT save and can be used in the ROOT [2] interpreter. The GCombiner class reference manual has been published as a HERA-B note, too [5].

2 Code and compilation

The GCombiner packages consists of the following files:

GCombiner.hh Class declaration file.

GCombiner.icc Class implementation file (included by GCombiner.hh)

As the GCombiner classes are fully templated, they can be used like STL classes by simply including the header file:

```
// This is file myappl.C
#include "GCombiner.hh"
```

The GCombiner classes are based on STL and need therefore a working installation of STL. GCombiner will be distributed together with the HERA-B analysis framework `BEE`.

3 A simple example

The functionality of the GCombiner is demonstrated in the following by using STL `vector<int>` containers:

```
// this is main.C
#include <iostream>
#include <vector>
#include "GCombiner.hh"

int main(void) {

  // to shorten the notation
  typedef vector<int> Int_v;

  // define vectors
  Int_v a(3);
  Int_v b(3);

  a[0] =  0, a[1] =  1, a[2] =  2;
```

```
  b[0] = 10, b[1] = 11, b[2] = 12;

  // to shorten the notation
  typedef GCombiner<Int_v>           GC_v;
  typedef GCombiner<Int_v>::iterator GC_it;

  // define the combiner:
  GC_v Comb;

  Comb.push_back(a);
  Comb.push_back(b);

  const GC_it end = Comb.end();
  for(GC_it it = Comb.begin(); it != end; ++it) {
    cout << " it: " << it << endl;
  }

  return 0;
}
```

The output of the code is:

```
it: 0 10
it: 1 10
it: 2 10
it: 0 11
it: 1 11
it: 2 11
it: 0 12
it: 1 12
it: 2 12
```

This shows the way how the objects from the different containers are combined.

In a real world example, a data analysis program might have provided us with a set of particle candidates. For each particle type we have a STL container which stores the pointers of the candidate tracks:

```
...

// to shorten the notation
typedef vector<RecoTrack*>  RecoTrack_v;

// define track containers
RecoTrack_v pi_plus  = ...; // fill them somehow
RecoTrack_v pi_minus = ...;

// define a combiner for track vectors
typedef GCombiner<RecoTrack_v>           GCombiner;
typedef GCombiner<RecoTrack_v>::iterator GCombiner_it;

GCombiner combiner;
```

```
// add vectors to combiner
combiner.push_back(pi_plus);
combiner.push_back(pi_minus);

// now combine pi_plus & pi_minus
for(GCombiner_it it = combiner.begin();
    it != combiner.end(); ++it) {
  ... // do something, e.g. vertexing,
      // for more details see below
}
```

The combiner can be used to combine objects from `ArteTable`'s:

```
#include "arte/arte.hh"
...

// to shorten the notation
typedef GCombiner<ArteTable<RTRA> >           GC_v;
typedef GCombiner<ArteTable<RTRA> >::iterator GC_it;

// define the combiner:
GC_v Comb;

// build combiner for RTRA tripletts:
Comb.push_back(ArteTable<RTRA>());
Comb.push_back(ArteTable<RTRA>());
Comb.push_back(ArteTable<RTRA>());

const GC_it end = Comb.end();
for(GC_it it = Comb.begin(); it != end; ++it) {
  // do something with RTRA triplett combinations
}
```

4 Using GCombiner

The GCombiner can only work on containers which fulfill the following conditions:

1. all containers `T` must be of same type (e.g. `vector<int>`)
2. container has a `T::iterator` typedef (e.g. `vector<int>::iterator`)
3. container has a `begin()` method (e.g. `vector<int>::begin()`)
4. container has an `end()` method (e.g. `vector<int>::end()`)
5. `T::iterator` has an `operator*()` defined
6. `T::iterator` has an `operator==()` defined
7. `T::iterator` has an `operator!=()` defined
8. `T::iterator` has an `operator++()` defined

This is the case for all STL containers, BEE `GObjArray` and `GClonesArray` containers and `ArteTable` containers.

In the following some details of the GCombiner functionality will be demonstrated.

4.1 Adding more containers

By using the `push_back()` method of the GCombiner, more containers can be added. In case of our first example one could easily build tripletts of `int` by adding another container:

```
// this is main.C
...
int main() {
  ...
  Int_v c(2);
  ...
  c[0] = 20;
  c[1] = 21;
  ...
  Comb.push_back(c);

  const GC_it end = Comb.end();
  for(GC_it it = Comb.begin(); it != end; ++it) {
    cout << " it: " << it << endl;
  }
}
```

Now one would get the following output:

```
it: 0 10 20
it: 1 10 20
it: 2 10 20
it: 3 10 20
it: 0 11 20
it: 1 11 20
it: 2 11 20
it: 3 11 20
it: 0 12 20
it: 1 12 20
it: 2 12 20
it: 3 12 20
it: 0 10 21
it: 1 10 21
it: 2 10 21
...
```

4.2 Accessing combination elements

A single combination is represented by the `GCombinationIterator` class. The `GCombinationIterator` itself is a container for iterators of the containers which are added to the GCombiner. Therefore a `GCombination-`

`Iterator` can never be a STL compatible iterator. The `GCombination-Iterator` class has iterator properties (like the `operator++()` method) as well as container properties. Those are used to access the elements of a combination.

In our example from section 4.1 we can access the elements of a combination via `operator[]()`:

```
// This is file main.C
...
  // to shorten notation
  typedef vector<int>::iterator  Int_it;

  // loop over combinations
  const GC_it end = Comb.end();
  for(GC_it it = Comb.begin(); it != end; ++it) {
    // get current iterator of container a:
    Int_it aiter = it[0];
    // get current iterator of container b:
    Int_it biter = it[1];
    // get current iterator of container c:
    Int_it citer = it[2];

    // output of combination:
    cout << " it: size: " << it.size() << " ["
         << *it[0] << ","
         << *it[1] << ","
         << *it[2] << "]" << endl;
  }
...
```

Note: The direct access via `operator[]()` does not provide a boundary check. The user has to assure that boundaries are not violated.

Another possibility is to loop over the elements of a single combination via an iterator:

```
// This is file main.C
...
// to shorten notation
typedef vector<int>  Int_v;

typedef GC_trait<Int_v>::GCombination_cit GCombination_cit;

// loop over combinations
const GC_it end = Comb.end();
for(GC_it it = Comb.begin(); it != end; ++it) {

  // loop over elements
  for(GCombination_cit cit = it.begin(); cit != it.end();
      ++cit)
    cout << **cit << " ";
  cout << endl;
}
```

```
...
```

It is most convenient to get the type of the element-iterator via the `GC_trait` like in the example above (for more information about the `GC_trait` see section 4.4).

Note: Although `GC_it` is **not** a STL compatible iteratator, `GCombination_cit` is STL compatible (see section 4.4)!

A third possibility is to get the container iterators by calling `GCombinerIterator::operator*()`:

```
// This is file main.C
...
// to shorten notation
typedef vector<int>                           Int_v;
typedef GC_trait<Int_v>::GCombination     GCombination;
typedef GC_trait<Int_v>::GCombination_cit GCombination_cit;

// loop over combinations
const GC_it end = Comb.end();
for(GC_it it = Comb.begin(); it != end; ++it) {

  // get reference to container of iterators
  const GCombination& iters = *it;

  // loop over elements
  for(GCombination_cit cit = iters.begin();
      cit != iters.end(); ++cit)
    cout << **cit << " ";
  cout << endl;
}
```

4.3 Selecting combinations

It might happen that not every combination is valid in some sense. In case one combines tracks one could want to avoid that the same track appears more than once in a track combination. Even more complicated, one could want to avoid that track clones[1] appear in a track combination.

To detect such an incompatibility between elements of a combination, the user can define his own "collision" function. A default collision function is provided by the GCombiner package. The collision function detects if the same object (according to the `operator==()` method) appears more than once in a combination:

```
// This is file main.C
... // see above examples

// to shorten the notation
typedef vector<int> Int_v;

// define vectors
```

[1]A track clone is a track which shares one or more of it's segments with other tracks.

```
  Int_v a(3), b(3);

  a[0] = 0, a[1] = 1, a[2] = 2;
  b[0] = 0, b[1] = 1, b[2] = 2;

  ... // see examples above

  // loop over combinations
  const GC_it end = Comb.end();
  for(GC_it it = Comb.begin(); it != end; ++it) {

    // skip combinations with identical elements
    if(collision(it)) continue;

    cout << " it: " << it << endl;
  }
```

The output of the code would look like this:

```
 it: 1 0
 it: 2 0
 it: 0 1
 it: 2 1
 it: 0 2
 it: 1 2
```

The `collision()` function can be specialized for any type of container. The fact that the collision detection is not an internal part of the GCombiner class design makes it very easy for the user to define his own selection routines. This improves the flexibility and extensibility of the GCombiner class, which is an important design property for reusable code and related to the so called "Open-Closed" principle [4].

The definition of the `collision()` function is a good starting point to write other collision functions:

```
template <class T>
bool collision(GCombinerIterator<T>& rhs) {
  typedef GC_trait<T>::GCombination_cit GCombination_cit;

  const GCombination_cit vend = rhs.end();
  for(GCombination_cit cit = rhs.begin(); cit!= vend; ++cit) {

    GCombination_cit cit2 = cit;
    for( ++cit2; cit2 != vend; ++cit2 ) {
      // attention: we compare values, not iterators!
      if(**cit == **cit2) {
        return true;
      }
    }
  }
  return false;
}
```

4.4 Configure GCombiners

The `GC_trait` template configures the internal behaviour of the GCombiner. The traits technique is very useful to allow for a type dependent template configuration (see [3]). The `GC_trait` provides a set of `typedef`'s which can be used by the programmer to get type definitions without knowing about the details of the `GCombiner` implementation:

```
template <class T>
struct GC_trait {
  ...
  typedef typename T::iterator                     GIterator;
  ...
  typedef std::vector<GIterator>                   GCombination;
  typedef std::vector<GIterator>::iterator         GCombination_it;
  typedef std::vector<GIterator>::const_iterator   GCombination_cit;
  ...
};
```

The usage of the `GC_trait` structure is demonstrated in the examples above.

5 Using SCombiner

In addition to the `GCombiner` class the `SCombiner` class is provided to make combinations from objects of a single container. `SCombiner` can work on containers which have

1. a `T::begin()` and `T::end()` method
2. a `T::iterator` typedef
3. a `T::size()` method

The user can make use of the `SCombiner` by including the header file:

```
#include "SCombiner.hh"
```

As the `SCombiner` is a fully templated set of classes (see STL) no library is necessary. In contrast to `GCombiner`, the `SCombiner` class needs two template arguments as can be seen from the declaration:

```
template <class T, unsigned int X> class SCombiner
template <class T, unsigned int X> class SCombinerIterator
```

The first template argument specifies the type of the STL compatible container, the second argument gives the number of objects per combination.

5.1 Generate a Combiner object

Two possibilities exist to specify the combiner from which combinations should be generated:

```
// this is file main.C
#include "SCombiner.hh"
#include <vector>
```

```
int main(void) {

  // define a STL container of ints
  typedef vector<int> Int_v;

  Int_v vec;
  vec.push_back(0);
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  // doublett combinations of ints
  typedef SCombiner<Int_v,2> GS_v;

  // 1st possibility: combiner specified
  // as a constructor argument
  GS_v SComb(vec);

  // 2nd possibility: default constructor,
  // combiner specified by set_container()
  // member function
  GS_v SComb2;
  SComb2.set_container(vec);
```

The first method is preferrable as it is slightly more efficient.

5.2 Accessing objects in a combination

The iteration over the various combinations and accessing the combination elements is identical to the `GCombiner` class:

```
  // above example, continued

  GS_v::iterator send = SComb.end();
  for(GS_v::iterator sit = SComb.begin(); sit != send; ++sit) {
    cout << *sit[0] << "," << *sit[1] << endl;
    // or:
    // cout << sit << endl;
  }
}
```

The output would look like this:

```
0 1
0 2
0 3
1 2
1 3
2 3
```

As one can see, each combination occurs only once (in contrast to `GCombiner`).

In order to make tripplet combinations, define the `SCombiner` as follows:

```
// This is file main.C
... // define vector of int's, see above

// tripplett combinations of ints
typedef SCombiner<Int_v,3> GS_v;

GS_v SComb(vec);

GS_v::iterator send = SComb.end();
for(GS_v::iterator sit = SComb.begin(); sit != send; ++sit) {
  cout << sit << endl;
}
```

As one can see, only the `GS_v` typedef has changed! The output will be the following:

```
0 1 2
0 1 3
0 2 3
1 2 3
```

Objects from ArteTable's can be combined in the following way:

```
...

// doublett combinations of RTRA's
typedef SCombiner<ArteTable<RTRA>,2> GS_v;

GS_v SComb; // do not give ArteTable as argument!

GS_v::iterator send = SComb.end();
for(GS_v::iterator sit = SComb.begin(); sit != send; ++sit) {
  cout << sit << endl;
}
```

Note: With ArteTables it is sufficient to call the default `SCombiner` constructor[2]!

As `ArteTableIterator`'s point to `ArtePointer` objects, the output will be like this (the `ArtePointer`'s are printed):

```
[1] [2]
[1] [3]
[1] [4]
[1] [5]
[1] [6]
...
```

5.3 Accessing the complementary objects to a combination

To access objects which are complementary to the objects present in a given combination, one can use `getComplement()`. This function compares

[2]ArteTables are global objects and therefore directly accessible, in contrast to a local container object.

the objects from a given SCombinerIterator (i.e. a combination) with the objects from a given input container and stores the non matching objects into an output container. For example:

```
// this is file main.C
... //

  Int_v vec;
  vec.push_back(0);
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  Int_v complement;

  typedef SCombiner<Int_v,2> GS_v;
  GS_v SComb(vec);

  GS_v::iterator send = SComb.end();
  for(GS_v::iterator sit = SComb.begin(); sit != send; ++sit) {

    getComplement<Int_v,2>(sit,vec,complement);

    cout << sit << "; getComplement: ";
    for(int i = 0; i != complement.size(); ++i) {
      cout << complement[i] << " ";
    }
    cout << endl;

    complement.clear();
  }
```

The output will look like:

```
0 1 ; getComplement: 2 3
0 2 ; getComplement: 1 3
0 3 ; getComplement: 1 2
1 2 ; getComplement: 0 3
1 3 ; getComplement: 0 2
2 3 ; getComplement: 0 1
```

5.4 Configure SCombiner

The `SCombiner` classes are based on the `GC_trait` class, therefore `SCombiner` can be configured by modifying the traits class. See also section 4.4.

References

[1] Stepanov, A. and Meng Lee, The Standard Template Library, ANSI X3J16-94-0095 / ISO WG21-NO482

[2] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Ana lysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[3] Nathan C. Myers, Traits: a new and useful template technique, C++ Report, June 1995

[4] Robert C. Martin, The Open Closed Principle, C++ Report, Jan. 1996

[5] T.Glebe, GCombiner Class Reference Manual, HERA-B Note 01-002, Jan. 2001