STAR Offline Library Long Writeup

StAssociationMaker

User Guide and Reference Manual

Contents

I User Guide **1**

1 Introduction **2**

2 How to read this document **3**

3 Further documentation **3**

4 Getting StAssociationMaker Sources **3**

5 Multimaps & the Standard Containers **4**

6 How to use it: StMcAnalysisMaker and StAssociator.C **8**

6.1 StMcAnalysisMaker::Make() 9

7 Known Problems **11**

II Reference Manual **12**

8 Global Constants **13**

9 Class Reference **13**

9.1 StAssociationMaker 14

9.2 StMcParameterDB 14

9.3 StTrackPairInfo 15

Part I

User Guide

1 Introduction

StAssociationMaker[1] is a package that uses the functionality of StEvent and StMcEvent together in order to have reconstructed and Monte Carlo information handy. The package is meant to be called in a chain after both StEvent and StMcEvent have been loaded in a ROOT session for a given event. The aim of StAssociationMaker is to establish the relationships, based on certain criteria, between the reconstructed and Monte Carlo information. In this way, users can easily check whether a particular reconstructed hit or track came from a Monte Carlo object, and directly obtain the associated Monte Carlo hit or track so that an assessment of the quality of the data can be made.

[1] We will adopt the convention used in StEvent and write StMcEvent when we refer to the package and *StMcEvent* when we refer to the class.

2 How to read this document

This document is divided in two parts, a user guide and a reference manual. The first part is the main part, since although the package contains several classes, the only class that is designed for general use is *StAssociationMaker*. Apart from the usual methods provided for makers, this class has only a few additional methods for general use: the methods to access the multimaps. The navigation and use of StEvent and StMcEvent are described in their respective documentation. This manual will cover these methods, and how to use the package. The user guide will also provide an explanation of the StMcAnalysisMaker package. This additional package is basically a working example of how to do some basic histograms from the multimaps built by the Association Maker. The class reference will be kept short for the reasons described above. In addition, a section describing the main features of the *multimap* class will also be provided. Because StAssociationMaker relies so heavily on multimaps, giving a brief description of their idea and usage is a necessary item in the documentation.

It remains true that understanding the various examples certainly is the best way to get started. In this case, StMcAnalysisMaker is actually a working example, and many of the things discussed here can be seen working "live" in this package. To make sure that it is working, one needs just to run the macro *StAssociator.C* at the root4star prompt. More information will be given later on actually checking it out and modifying for your purposes 6.

3 Further documentation

StAssociationMaker makes use of various classes from the StarClassLibrary (SCL). To obtain the SCL documentation perform the following steps:

1. Obtain an *afs* token: *klog -cell rhic.*
2. Make sure *$CVSROOT* is set properly:
   (i.e. *CVSROOT = /afs/rhic.bnl.gov/star/packages/repository*)
3. Check-out the SCL into your current working directory:
   cvs checkout StRoot/StarClassLibrary
4. Go to the directory containing the documentation:
   cd StRoot/StarClassLibrary/doc/tex
5. Create the PostScript document *StarClassLibrary.ps*:
   make

4 Getting StAssociationMaker Sources

To access the complete source code proceed as follows:

StAssociationMaker is under **CVS** control at BNL. It can be accessed via *afs*:

1. Obtain an *afs* token: *klog -cell rhic.*
2. Make sure *$CVSROOT* is set properly:
   (i.e. *CVSROOT = /afs/rhic.bnl.gov/star/packages/repository*)
3. Check-out package into your current working directory:
   cvs checkout StRoot/StAssociationMaker
4. Check-out the examples *cvs checkout StRoot/StMcAnalysisMaker*
5. Copy the macro your current working directory:
   cp $STAR/StRoot/macros/examples/StAssociator.C /.

5 Multimaps & the Standard Containers

The definition of a multimap is not too long. A multimap [2] is a type of Standard Template Library associative container. that allows for duplicate keys. Now, initially, this definition doesn't say much, unless you're already a C++ guru and know exactly what each part of the definition means. So I'll try to explain it little by little. Then we'll see how it all falls together. The Standard Template Library (or STL) is a repository of several very useful classes that can make programming a whole lot easier. A few of the things provided by the Standard Library are *containers*, *algorithms*, *diagnostic* classes, *string* handling classes, *input/output* classes, and some others. As can be imagined, there is enough in the Standard Library to warrant several books worth of documentation, which I don't want to get into. There are quite a few tutorials on the web about it also. Part III of Stroustrup's C++ book is totally devoted to the Standard Library, which is an excellent starting point for more information.

The multimap is part of the STL, as one if the containers it provides. There are several containers with different features. Examples are *lists*, *vectors*, and associative arrays like *map* and of course *multimap*. The C++ standard library containers were designed to meet 2 criteria: to provide as much freedom as possible in the design of individual containers and at the same time to provide a common interface to users. This allows to make each container as efficient as possible for its intended use but still enable users to write code that is independent of the particular container being used. The standard library defines two kinds of containers: sequences and associative containers. A key idea for the standard containers is that they should be logically interchangeable wherever reasonable. Users can then choose between them depending on efficiency concerns and the need for specialized operations. For example, if lookup based on a key is common, a *map* can be used. If general list operations dominate, a *list* can be used. If many additions and removals of elements occur at the ends of the container, a *deque* (double-ended queue), a *stack*, or a *queue* should be considered.

A first idea of what a *map* is can be "a fast lookup table". You have entries like in a *vector* container. The difference is that every entry is a PAIR of things. That is, a map can be thought of as a collection of pairs of objects that are queried based on the first element of the pair (the KEY). The other element of the pair is referred to as the associated VALUE.

My favorite example to illustrate the concept in a more familiar context, is to think of a phonebook. The idea of a phonebook is that it contains people's phone numbers. Simple. Now, in the language we just learned,

[2] For reference look in Stroustrup's C++ book, 3d Ed. pp. 484-490

a phonebook is an associative container between a character string (a person's name) and an integer (the person's phone number). To use a phonebook, one looks up a person's name and reads off the associated phone number. The character string representing the name is the "key", and the phone number is the key's associated "value".

The difference between a multimap and a map is that the multimap allows for several entries having the same key, whereas in a map no 2 entries can have the same key. This means that if a phonebook were a *map*, then each person would only be able to have one phone number. Note that the phone number could be shared between people (only the keys are unique). A *multimap*, on the other hand, allows multiple keys. This means that if a phonebook were a *multimap*, then everyone could have as many phone numbers as they pleased.

One thing that help to picture the map and the multimap is the following. Recall the definition is that they are "associative containers of pairs of objects". You can always think of a vector as an array. So if you have a `vector<int>` then you can picture it like this:

Vector:	1	2	5	17	20	25

So the first element is 1, the second is 2 and so on. Now, a map is a container of PAIRS of objects. The 2 objects can be anything, (i.e. the PAIR class is also a template.) So a phonebook would be, for example, a

`map<string,int, less<string> >`

I would picture it like this:

Map:	
("Brian",2034322043)	⟵ `phonebook.begin()`
("Manuel",6313448342)	
("Thomas",2034325829)	
	⟵ `phonebook.end()`

Now, the 'key' here would be the `string`, and the 'value' the `int` representing the phone number. Since it is a map, keys are unique. This means that there will only be one entry for each string. If I say:

```
phonebook["Manuel"] = 2034325637;
```

the entry for "Manuel" will be overwritten.

Note that I can say

```
phonebook["Thomas"] = phonebook["Manuel"];
```

and I'll overwrite Thomas's phone number with mine. The 'values' can be repeated.

We see here that a map supports indexing. Since there is one entry for each key (and only one), it is unambiguous which entry one wants, so we can use the key as an index. A multimap doesn't support indexing, because there is still the ambiguity of having multiple entries. If we made the phonebook a

```
multimap<string, int, less<string> >
```

then we could have

Multimap:	
(”Brian”,2034322043)	
(”Manuel”,6313448342)	⟵ `phonebook.lower_bound(‘‘Manuel’’);`
(”Manuel”,2034325637)	
(”Thomas”,2034325829)	⟵ `phonebook.upper_bound(‘‘Manuel’’);`

Because simple subscripting by keys (like an array) is **not** supported for a multimap, the main way to access the multiple associated values given to a particular key is through iterators. Iterators can be thought of as a pointer to an element of a container which allow one to navigate through it. For example, for the *list* container, (or *vector* or pretty much any STL container) one can loop through all the elements by doing

```
list<int> myList;
// something is done and the list is filled

list<int>::iterator listIter;
for (listIter = myList.begin(); listIter != myList.end(); listIter++){
    // do something with the entry
}
```

The `begin()` method returns an iterator to the first element of the container, and the `end()` method returns an iterator to the last-plus-one element of the container. So to iterate through a container, we need 2 iterators because you don’t know a priori how many entries there are. One will be the starting point and the other the end point. For iterating through the entries with a common key, we use lower_bound and upper_bound. You can think of them as the equivalent to `vector.begin();` and `vector.end();`

The use of iterators to navigate through a sequence of entries with a similar key is possible because all the entries in the multimap are ordered according to some criterion. The entries with a similar key are then placed right next to each other in the container. In the phonebook example, all the entries for “Manuel” are placed next to each other. In the example, we took the default comparison for strings, which is just alphabetical order. That is where the `less<string>` part comes in, for the example phonebook. One has to specify a comparison, if the compiler doesn’t take default template arguments. And even if it does, for other user defined classes one has to provide a custom made ordering. This custom made ordering can also be implemented for a built in class, if need be. The bottom line is that the elements of the multimap with the same key end up next to each other, ready to be looped over with the iterators.

It is important to stress that every entry in the multimap is a key-value pair, regardless of whether the key has appeared before. This means that there is some redundancy. If you loop over a sequence of entries using lower_bound and upper_bound, ALL of the entries in that sequence will have the same key:

Multimap:	
(”Manuel”,631...)	⟵ first entry with key = “Manuel”
(”Manuel”,203...)	⟵ second entry, same **key**, different **value**
(”Manuel”,212...)	⟵ third entry, etc.

For a multimap, then, *lower_bound(k)* and *upper_bound(k)* give the beginning and end of the subsequence of elements of the map that have the key "k". However, as Stroustrup says, doing 2 operations all the time is neither elegant nor efficient. The operation *equal_range(k)* is provided to deliver both. For examples of this usage refer to sec. 6.

The multimaps used in StAssociationMaker keep a relationship between the reconstructed StEvent objects and Monte Carlo StMcEvent objects. This relationship is established starting from the hit information and building up from there.

The following multimaps are now implemented in StAssociationMaker:

- TPC Hits
- SVT Hits
- FTPC Hits
- Tracks (using all the previous hit multimaps)
- Kink Vertices
- V0 Vertices
- Xi Vertices

The association is made based on criteria given by the user. These criteria are established at runtime, and the user controls them at the macro level.

The criterion for associating a given reconstructed hit to a Monte Carlo hit is based on their spatial proximity, in the global coordinate system.

Once the hit associations are made, the criterion for associating a reconstructed track to a Monte Carlo track is based on the number of hits they have in common. This means that to build the Track Multimap, the Hit Multimaps are used. The defaults are:

```
TPC Cuts
X Cut    : 5 mm
Y Cut    : 5 mm
Z Cut    : 2 mm
Required TPC Hits for Associating Tracks : 3
SVT Cuts
X Cut    : 1 mm
Y Cut    : 1 mm
Z Cut    : 1 mm
Required SVT Hits for Associating Tracks : 1
FTPC Cuts
R Cut    : 3 mm
Phi Cut  : 5 degrees
Required FTPC Hits for Associating Tracks : 3
```

These numbers are supposed to be much larger than the intrinsic resolution of the detectors. The idea is to keep these numbers loose so the map would really rule out bad associations but keep enough objects in the multimap to make further analysis downstream.

In addition, this version of StAssociationMakeralso provides 2 multimaps for each of the objects above. That is, there are 2 multimaps for TPC Hits, 2 multimaps for SVT Hits, and so on. One of the multimaps uses the **reconstructed** objects from StEvent as keys, and the other multimap uses the **Monte Carlo** objects from StMcEvent as keys. Even though this might appear redundant at first (there is a one to one correspondence between the entries in the reconstructed map and the entries in the monte carlo map) remember that one can only lookup entries based on the key. There are cases where we might want to lookup based on the reconstructed objects, and there are cases where we will want to use the monte carlo objects. Having 2 maps allows to find an association in both directions.

Almost all multimaps are just between a reconstructed object and a monte carlo object. The notable exception are the Track multimaps. For the reconstructed track multimap, instead of just associating an *StGlobalTrack* to an *StMcTrack*, we make the association between an *StGlobalTrack* and an instance of an *StTrackPairInfo* class. This class has, of course, a pointer to the associated *StMcTrack* but in additon has a member that keeps track of the number of hits the 2 tracks have in common for each of the detectors, and member functions that return the *percent* of hits in each detector that are associated out of the total number of hits the track has. This is done so that later on one can make more stringent cuts on the tracks. If any additional information is needed, we would only need to modify the *StTrackPairInfo* class to accomodate it. For the monte carlo track multimap, the "value" of the multimap is also the same pointer to the *StTrackPairInfo* class. This means, of course, that *StTrackPairInfo* must also have a pointer to the associated *StGlobalTrack*, in addition to the *StMcTrack*.

We said that the default values of required hits in common, spatial proximity, etc. can also be modified at the macro level. The cuts are kept in an instance of the singleton class *StMcParameterDB*. This class is instantiated exactly once (hence the name "singleton") and the rest of the classes in StAssociationMaker look there to figure out the cut criteria. For an example of how to change the cuts, look in the *StAssociator.C* macro.

6 How to use it: StMcAnalysisMaker and StAssociator.C

The procedure starting from scratch to run the provided example is just to run the macro in any directory. The package should run in any platform and any version of the library.

```
ssh rcas6001
starnew
mkdir workdir
cd workdir
root4star
.x StAssociator.C
```

To get the code you can just check it out from the library. Probably the one you'll want to check out and modify is StMcAnalysisMaker, since it is filled with examples. But one can certainly check out the

Association Maker to look at the source, which is always the most up to date documentation of what is available.

```
cvs co StRoot/StAssociationMaker
cvs co StRoot/StMcAnalysisMaker
```

You can edit the Analysis maker and then compile using cons.

```
cons +StMcAnalysisMaker/
cvs co StRoot/macros/examples/StAssociator.C
Edit the macro to use your maker, if you wisely renamed it before editing.
root4star StAssociator.C
```

Note that to compile using cons, you don't need to give the exact name of the maker. A substring works, but it will try to compile everything that matches the substring. This means that if you have several packages checked out and you want to compile them all, you can probably do

```
cons +St
```

and this will do the trick.

StAssociator.C can be found in *$CVSROOT/StRoot/macros/examples/StAssociator.C*. To get it you must use `cvs co` command as above. The macro runs a chain consisting of four makers:

***StEventReaderMaker*:** loads StEvent

***StMcEventMaker*:** loads StMcEvent

***StAssociationMaker*:** creates the hit and track associations

***StMcAnalysisMaker*:** Picks up the previous info. and analyzes it (incorporates a few simple examples)

At the moment, it runs the chain on a ROOT file tree. That is, it uses the DST and GEANT braches of the files it finds in the directory. Example invocation is

```
.x StAssociator.C(10,
    "/disk00000/star/test/new/tfs_Solaris/year_2a/psc0210_01_40evts.geant.root")

   processes 10 events from the specified file
```

It takes the path to look for files from the specified file, but the files it actually opens depend on what branches are activated in the macro. This is done inside the macro with the command *SetBranch*.

6.1 StMcAnalysisMaker::Make()

We will explain here the first example in the *Make()* method of *StMcAnalysisMaker*, since this is the part that most users will follow. Recall that when *StMcAnalysisMaker::Make()* is called, the following should have already happened:

6.1 StMcAnalysisMaker::Make()

1. StEvent should be loaded in the chain.
2. StMcEvent should be loaded in the chain.
3. StAssociationMaker should have finished making associations.

We need to access the packages and information that is already loaded. This is done by getting a pointer to *StEvent*, a pointer to *StMcEvent*, and a pointer to *StAssociationMaker*:

```
// Get the pointers we need, we have to use the titles we gave them in the
// macro.  I just used the defaults.
StEvent* rEvent = 0;
rEvent = ((StEventMaker*) gStChain->Maker("events"))->event();
StMcEvent* mEvent = 0;
mEvent = ((StMcEventMaker*) gStChain->Maker("MCEvent"))->currentMcEvent();
StAssociationMaker* assoc = 0;
assoc = (StAssociationMaker*) gStChain->Maker("Associations");
```

The pointer to *StAssociationMaker* then enables us to get the pointers to the appropriate multimpas:

```
tpcHitMapType* theHitMap = 0;
theHitMap = assoc->tpcHitMap();
trackMapType* theTrackMap = 0;
theTrackMap = assoc->trackMap();
```

The classes *tpcHitMapType* and *trackMapType* are the typedef's to the appropriate multimaps. We see here that to get a the pointers to them we just need to call the *StAssociationMaker::tpcHitMap()* and *StAssociationMaker::trackMap()* methods.

Once we have the pointers to the maps, we can start analyzing them. The first example in the maker shows how to check whether a particular reconstructed TPC Hit was associated with a Monte Carlo hit.

```
// Example: look at hits associated with 1st REC hit in Tpc Hit collection.

StTpcHit*     firstHit;
firstHit = *( rEvent->tpcHitCollection()->begin() );
cout << "Assigned First Hit: " << endl;
cout << *firstHit << endl;
cout << "This hit has " <<  theHitMap->count(firstHit);
cout << " MC Hits associated with it."<< endl;

// To get the associated hits of the first hit we use equal_range(key),
```

```
// which returns 2 iterators, the lower bound and upper bound,
// so that then we can loop over them.

cout << "Position of First Rec. Hit and Associated (if any) MC Hit:" << endl;
pair<tpcHitMapIter,tpcHitMapIter> hitBounds = theHitMap->equal_range(firstHit);

for (tpcHitMapIter it=hitBounds.first; it!=hitBounds.second; ++it) {
   cout << "[" << (*it).first->position();
   cout << ", "<< (*it).second->position() << "]" << endl;
}
```

The other examples in the maker show

1. how to make a histogram using the Hit multimap
2. how to use an *StGlobalTrack* to get its partner in the Track multimap.
3. how to make a histogram using the Track multimap

(Note that the histograms were defined in the header file as public data members.)

To play with it yourself you can pick up StMcAnalysisMaker and use it as a template for a Maker of your own that works with StMcEvent:

```
mkdir StRoot/StMyMcAnalysisMaker
cp  $STAR/StRoot/StMcAnalysisMaker/* StRoot/StMyMcAnalysisMaker/
[edit]
makel -C StRoot/StMyMcAnalysisMaker
cvs co $CVSROOT/StRoot/macros/examples/StAssociator.C  (and edit to use your make
root4star StAssociator.C
```

7 Known Problems

StAssociationMaker is not yet complete. It has only been tested in new, SL99e. We do not foresee to make it work in dev (SL99f) since StEvent itself is going through a major overhaul. The next version will be developed once the new StEvent is in place, and also the changes in the g2t_vertex table have been implemented.

The FTPC and SVT associations between reconstructed and Monte Carlo hits are not included presently (although the appropriate Collections are filled by *StMcEventMaker*). Also, StAssociationMaker does not compile with SUN CC4.2 yet, because that compiler has problems handling multimaps, and this is being worked on.

Part II

Reference Manual

8 Global Constants

We use the constants defined in the two header files *SystemOfUnits.h* and *PhysicalConstants.h* which are part of the StarClassLibrary. The types defined therein are used throughout StMcEvent .

9 Class Reference

The classes which are currently implemented and available from the STAR CVS repository are described in alphabetic order. Only the classes normally accessible to users are described.

9.1 StAssociationMaker

Summary *StAssociationMaker* is the main class in the StAssociationMaker package. It aim is to create the multimaps, and to provide methods to access them.

Synopsis
```
#include "StAssociationMaker.h"
class StAssociationMaker;
```

Description Objects of type *StAssociationMaker* are the entry point to the hit and track multimaps. From here one can navigate to (and access) quantities stored in them. Of course, one normally needs to include the necessary StEvent and StMcEvent classes to use in one's own analysis maker.

Persistence None

Related Classes None

Public Constructors

`StAssociationMaker(const char* name = "Associations",const char* title =`

Maker constructor.

Public Member Functions

`virtual void Clear(const char* opt="");`
Deletes the multimaps and the single histogram data member. Then calls standard Clear() for makers.

`virtual Int_t Init();`
Standard for makers.

`virtual Int_t Make();`
Makes hit and track associations, i.e. builds the multimaps.

`virtual Int_t Finish();`
Standard for makers.

`tpcHitMapType* tpcHitMap();`
Returns the pointer to the TPC Hit multimap. This is the way to get access to the map from another maker. Look in sec. 6

`trackMapType* trackMap()`
Returns the pointer to the Track multimap. This is the way to get access to the map from another maker. Look in sec. 6

9.2 StMcParameterDB

Summary *StMcParameterDB* is a singleton class that keeps track of the cuts to be used for making hit and track associations. If used, it should be called from the macro, so an include command should not be necessary since it is already included in *StAssociationMaker*.

Synopsis

```
#include "StMcParameterDB.hh"
class StMcParameterDB;
```

Description

To use it in the macro one just needs to get a pointer to the only instance of the class.
This is done through the StMcParameterDB::instance() method: `StMcParameterDB* parameter`

One then uses this pointer to access the methods of the class.

Persistence

None

Related Classes

None

Public Constructors

None

The class only has private constructors. The only access to it is through its member functions.

Public Member Functions

`static StMcParameterDB* instance();`
Returns a pointer to the only instance of the class. This is the way to access the the class from another macri. Look in *StAssociator.C* for an example.

`double xCut() const;`
Returns the present value of the cut in the x direction for hit associations.

`double zCut() const;`
Returns the present value of the cut in the z direction for hit associations.

`unsigned int reqCommonHits() const;`
Returns the present value of the cut in the number of common hits for track associations.

`void setXCut(double);`
Sets the value of the cut in the x direction for hit associations.

`void setZCut(double);`
Sets the value of the cut in the z direction for hit associations.

`void setReqCommonHits(unsigned int);`
Sets the value of the cut in the number of common hits for track associations.

9.3 StTrackPairInfo

Summary

StTrackPairInfo is the class that contains the information of the pair of a particular *StGlobalTrack*. It has as members the associated *StMcTrack* and the number of common hits beween the paired tracks.

Synopsis

```
#include "StTrackPairInfo.hh"
class StTrackPairInfo;
```

Description *StTrackPairInfo* provides a way to keep the information gained during the building of the multimap, like the number of common hits between tracks, for later use in the analysis.

Persistence None

Related Classes This class related to *StMcTrack* through composition, i.e. it contains a pointer to an *StMcTrack* as a member.

Public Constructors

```
StTrackPairInfo(StMcTrack* t, unsigned int pings);
```
Constructs an instance of *StTrackPairInfo* with partner MC Track *t* and number of common hits *pings*.

Public Member Functions

```
StMcTrack* partnerMcTrack() const;
```
Returns a pointer to the *StMcTrack* paired with a particular *StGlobalTrack* used as a key in the Track map.

```
unsigned int commonHits() const;
```
Number of common hits between the paired tracks.

```
void setPartnerMcTrack(StMcTrack*);
```
Sets the partner track. Should not normally be used, since this is taken care of during construction of the class.

```
void setCommonHits(unsigned int);
```
Sets the number of common hits. Should not normally be used, since this is taken care of during construction of the class.

Index

A
afs 3

B
begin() 6

C
Clear() 14
cons 9
CVS 3
CVSROOT 3

E
end() 6
event header 14

F
Finish() 14

G
Getting StAssociationMaker sources 3

H
Hit association 7

I
Init() 14
instance() 15
iterators 6

K
KEY for a multimap 4
known problems (StAssociationMaker) 11

M
Make() 14
multimap 4

R
reqCommonHits() 15
ROOT files 8
root4star 8

S
SCL 3
StarClassLibrary 3, 13
StAssociationMaker 2, **14**
StAssociator.C 8
StMcAnalysisMaker 8
StMcParameterDB **14**
StMcTrack 16
StTrackPairInfo 8, **15**

T
tpcHitMap() 14
tpcHitMapType 10
Track association 7
trackMap() 14
trackMapType 10

X
xCut() 15

Z
zCut() 15