Modular Software Development for MarlinReco et al.

Frank Gaede

ILCSoft Meeting

DESY, May 03, 2006

Outline

- Coding conventions
  - Style guidelines
- STL classes
  - container and algorithms
- Software philosophy
  - the LCIO-Marlin Paradigm
- Modular programming
  - example NN-Clustering

Style guidelines - Introduction

- coding style guidelines are recommendations with the aim to:
  - enhance the readability of code developed for one project or within one group
  - thus increase understanding of code written by one self and others
  - and improve maintainability and quality of code
- feel free to violate any rule of the guidelines
  - in order to enhance readability
  - if you have strong personal objections against a specific rule
  - if you think you can convince QA of your team
- **coding style guidelines are there to help you and not to serve as reasons for religious wars !**

variable names start with lower case

- variable names are mixed case starting with lower case but every following word with upper case
  - SavingsAccount someAccount ;
  - const ColMap* collectionMap ;
- the larger the scope the longer (explicit) the variable name:
  - static SimpleCluster* largestEnergyClusterInEvent ;
  - int nClu = col->getNumberOfElements() ; // local helper variable
- loop variables typically are: i,j,k,l,m,n:
  - for(int i=0; i< nClu ; i++){ ... }

type names start with upper case

- type names are mixed case starting every word and the type name itself with upper case:
  - class SavingsAccount ;
  - typedef std::map< std::string, lcio::LCCollection> ColMap ;
  - struct SimpleCluster ;
  - class TrackClusterLink ;

methods/functions start with lower case

- method/function names are verbs with mixed case starting with lower case but every following word with upper case
  - float SavingsAccount::getBalance() ;
  - SimCalorimeterHit::addMCParticleContribution(...) ;
- typically simple attributes are accessible via a get/set pair of methods:
  - float getEnergy() / void setEnergy(float energy)
- boolean attributes start with 'is' or 'has' :
  - isBackScatter() , hasEndPoint(), isFinalState() ,...

constants are all upper case

- constants are all upper case – optionally using '_' to separate words:
  - M_PI , LCIO::SIMTRACKERHIT , LCIO::WRITE_NEW
- prefer real constants (static class members defined in declaration) to #defines
- -> public class member variables should always be constant and all upper case

member variables start with '_'

- all member variables (protected and private) are prefixed with '_' in order to clearly distinct those in code from local variables

```
class MCParticle{
//...
protected:
  double _energy ;
  double _momentum[3] ;
  double _mass ;
};
```

make casts explicit

- C-style casts are discouraged and only allowed for basic types – use dynamic_cast, static_cast and reinterpret_cast instead:
  - float pi = (float) M_PI ; // **OK**
  - SimCalorimeterHit* hit =
  - (SimCalorimeterHit*) col->getElementAt(i) ; // **NO !**
  - dynamic_cast<SimCalorimeterHit*>(col ...) ; // **YES !**

use reasonable indentation

- use indentation of 2,3 or 4 characters to emphasize the logical layout, (e.g. use emacs c++-mode)

```
while( !isOver ){
   isOver = keepDoingSomething() ;
}

if( conditionA ){
   doSomething() ;
} else {
   if( conditionB ){
      doNothing() ;
   }
}
```

don't squeeze the code

- use empty lines and whitespace to enhance the readability of the code

```
for(int i=0;i<nClu;i++){                    // somewhat squeezed
   doTheLoopThing() ;
}

for( int i = 0 ; i < nClu ;  i++ ) {        // better  :-)

   doTheLoopThing() ;
}
```

document the code

- code should be documented using javadoc/doxygen documentation style:

```
/** A class that does the following for some good reason.
 *  Describe the main purpose and use cases here.
 *
 *  @author F.Gaede, DESY
 *  @version $Id: $
 */
class SomeNewClass { ...
} ;
```

- document at least: classes and public member functions
- use standard C++ comments for comments for developers in code
  - // here we need to transform into the CMS
  - vCMS = vLab.boost(cmsVector) ;

stick to the standard

- **use only ANSI C++ and STL**
  - for compiling use. e.g.

    g++ -c -Wall -ansi -pedantic
  - modify code until NO warnings persists

actively use CVS

- use CVS to manage changes to the source code
- frequently check in changes – documented and tagged (daily !)
- only check in code that compiles
  - if you need to make larger changes create a **branch** and develop in this branch until you are ready to merge the new development into the main
- check in only in your package subdirectory
- communicate with your colleagues in case of doubt
- what's not in CVS hasn't been done, yet !

some general remarks

- keep it simple:
  - good programs can be read and understood by any of your colleagues (and yourself after 6 month)
  - complicated code is more error prone and harder to debug
- don't spend to much thought on optimizing the code for CPU performance:
  - the optimizer is usually better than you think
  - most code is executed only a small number of times/run anyway
  - optimize only critical (nested) loops after they have been identified
- test, test and test your code:
  - provide some simple (documented) example/test programs for your code
- read other code that exists within the group to get some ideas on how things can be done
- prefer STL algorithms over self written code
- **keep it simple !**

Standard Template Library

- C++ comes with a very powerful set of templates that make your life easier: the **STL**
- at the heart of STL are two types of classes/functions:
  - containers (and *iterators*)
  - algorithms
- STL has been developed by the real experts
  - it is highly optimized
  - thoroughly tested
  - standardized
  - makes your code more readable and efficient

STL containers

- vector
  - one dimensional array
- list
  - doubly linked list
- deque
  - doubly ended queue
- set, queue, stack,...
- map
  - associative array (set of key value pairs)
- *pair*

STL algorithms

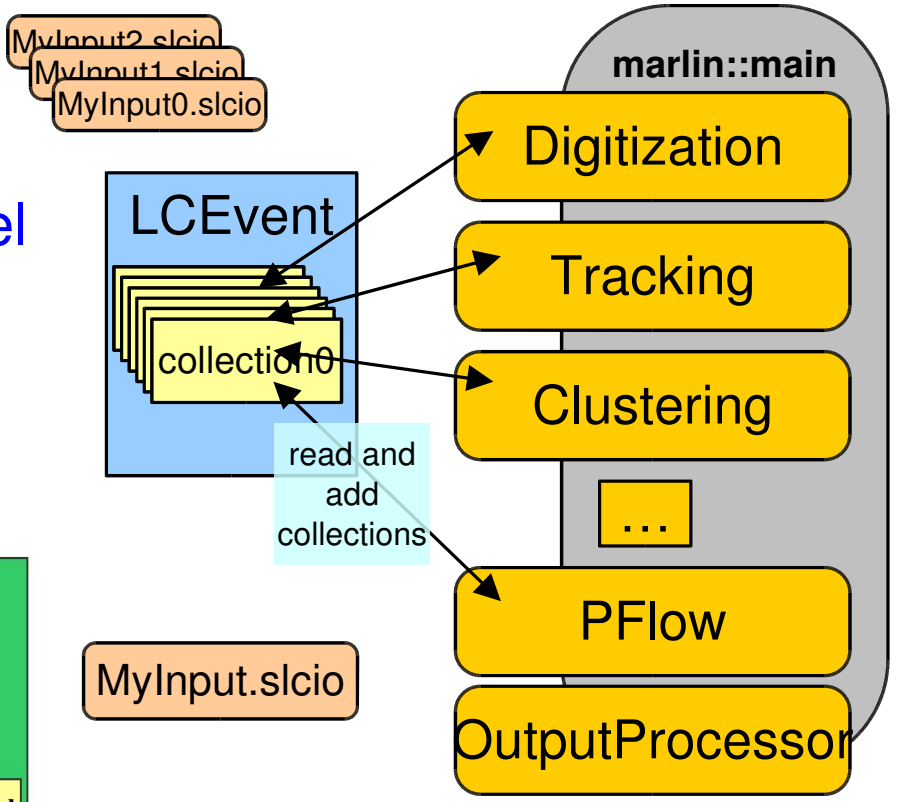
- typically work with all STL containers through the use of iterators, e.g.
  - std::transform(cl.begin(), cl.end(), std::back_inserter (*lcioClusters) , converter) ;
- copy
- sort
- find
- transform

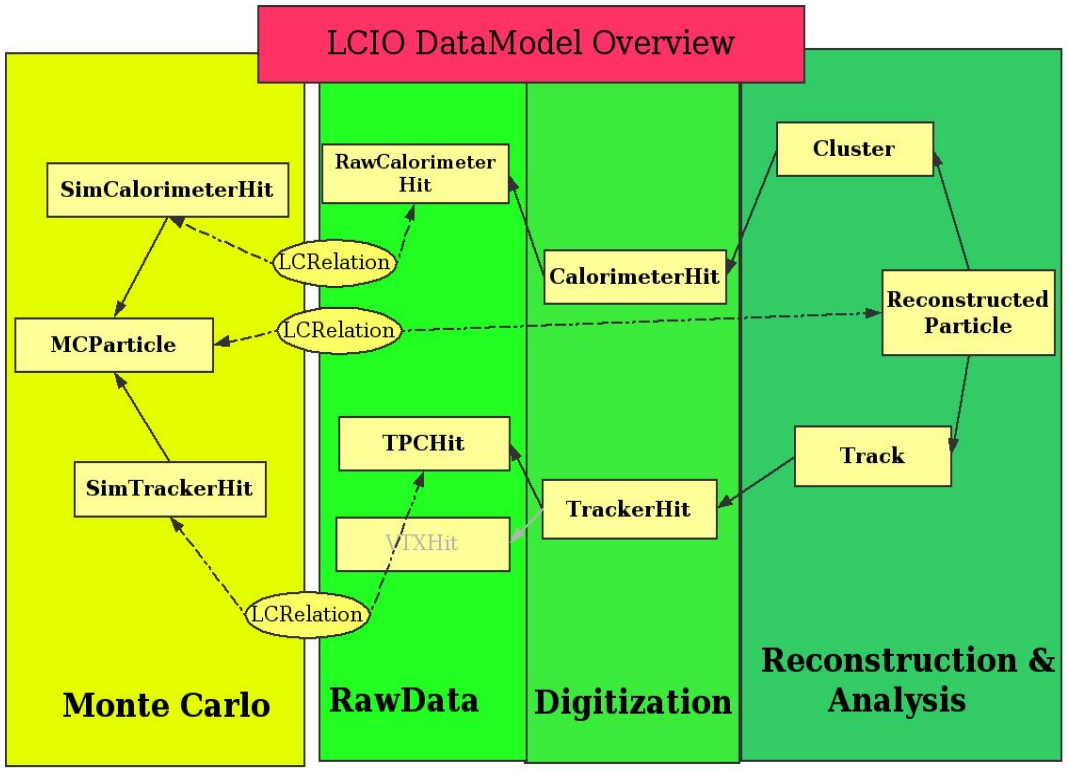
LCIO -Marlin SW paradigm

- keep data structures and algorithms separated !
- LCIO defines the data model that is used in ILC computing
- Marlin provides the **modular** framework for algorithms that operate on the data

Marlin/LCIO

- modular C++ **application framework** for the analysis and reconstruction of LCIO data
- uses LCIO as transient data model
- software modules called Processors
  - **Plug&Play** of processors





Module (wikipedia.org)

- In computer science, a module is a software entity that groups a set of (typically cohesive) subprograms and data structures. Modules are units that can be compiled separately, which makes them reusable and **allows multiple programmers to work on different modules simultaneously**. Modules also promote modularity and encapsulation (i.e. information hiding), both of which can make complex programs easier to understand.
- Modules provide a separation between **interface** and **implementation**.

Algorithms
Marlin processors
MarlinReco package

LCIO data model
(+extensions?)

modular PFA development

- proposal to develop PFA in a modular way:
  - write your algorithm in terms of *abstract* classes and function
  - use STL containers and algorithms
  - use a well defined interface for input and output – preferably LCIO collections of LCIO objects
  - provide one or several Marlin processors that allow to
    - run your algorithm in a Marlin program
    - produces check plots (AIDA histograms)
    - uses only trivial code (no algorithmic part)
    - serves as example for others on how to use your code

example NNClustering

- the next slides show an example of a generic module (package) that does NN type clustering

generic hit

```
template <class U>
class GenericCluster ;

/** Generalized hits points back to cluster, templated with original hit class.
 */
template <class T>
class GenericHit : public  std::pair< T*, GenericCluster<T>* >{

  typedef T value_type ;

public:

  GenericHit(T* hit, int index0 = 0 ) : Index0( index0 )   {
    first = hit ;
    second = 0 ;
  }

  GenericHit(T* hit ,  GenericCluster<T>* cl , int index0 = 0) : Index0( index0 ) {
    first =  hit ;
    second = cl ;
  }

  /** Index that can be used to code nearest neighbour bins, e.g. in z-coordinate
   *  to speed up the clustering process.
   */
  int Index0 ;
} ;
```

generic cluster

```
/**Generalized cluster - holds list of GenericHits, templated with original hit class.
 */
template <class T >
class GenericCluster : public std::list< GenericHit<T> * > {

public :

   GenericCluster( GenericHit<T>* hit) {
     addHit( hit ) ;
   }

  void addHit( GenericHit<T>* hit ) {

    hit->second = this ;
    push_back( hit ) ;

  }

  void mergeClusters( GenericCluster<T>* cl ) {

    for( typename GenericCluster<T>::iterator it = cl->begin() ; it !=  cl->end() ; it++ ){
      (*it)->second = this  ;
    }
    merge( *cl ) ;
  }

} ;
```

generic NN clustering

```
template <class In, class Out, class Pred >
void cluster( In first, In last, Out result, Pred* pred ) {

    typedef typename In::value_type GenericHitPtr ;
    typedef typename Pred::hit_type HitType ;

    typedef std::vector< GenericCluster<HitType >* >  ClusterList ;

    ClusterList tmp ;
    tmp.reserve( 256 ) ;

    while( first != last ) {

        for( In other = first+1 ; other != last  ; other ++ ) {

            if( pred->mergeHits( (*first) , (*other) ) ) {

                if( (*first)->second == 0 && (*other)->second == 0 ) {  // no cluster exists

                    GenericCluster<HitType >* cl = new GenericCluster<HitType >( (*first) ) ;

                    cl->addHit( (*other) ) ;

                    tmp.push_back( cl ) ;

                }
                else if( (*first)->second != 0 && (*other)->second != 0 ) { // two clusters

                    (*first)->second->mergeClusters( (*other)->second ) ;

                } else {  // one cluster exists

                    if( (*first)->second != 0 ) {
                        (*first)->second->addHit( (*other)  ) ;
                    } else {
                        (*other)->second->addHit( (*first)  ) ;
                    }
                }
            } // dCut
        }
        ++first ;
    }
    // remove empty clusters
```

NN distance functor class

```
/** Simple predicate class for nearest neighbour clustering. Requires
 *  PosType* HitClass::getPosition(), e.g for CalorimeterHits use: <br>
 *  NNDistance<CalorimeterHit,float> dist( myDistCut ) ;
 */
template <class HitClass, typename PosType >
class NNDistance{
public:

  /** Required typedef for cluster algorithm
   */
  typedef HitClass hit_type ;

  /** C'tor takes merge distance */
  NNDistance(float dCut) : _dCutSquared( dCut*dCut ) {}


  /** Merge condition: true if distance  is less than dCut given in the C'tor.*/
  inline bool mergeHits( GenericHit<HitClass>* h0, GenericHit<HitClass>* h1){

    if( std::abs( h0->Index0 - h1->Index0 ) > 1 ) return false ;

    const PosType* pos0 =  h0->first->getPosition() ;
    const PosType* pos1 =  h1->first->getPosition() ;

    return
      ( pos0[0] - pos1[0] ) * ( pos0[0] - pos1[0] ) +
      ( pos0[1] - pos1[1] ) * ( pos0[1] - pos1[1] ) +
      ( pos0[2] - pos1[2] ) * ( pos0[2] - pos1[2] )
      < _dCutSquared ;
  }

protected:
  NNDistance() ;
  float _dCutSquared ;
} ;
```

NNClusterProcessor

```
void NNClusterProcessor::processEvent( LCEvent * evt ) {

  clock_t start =  clock () ;


  LCCollectionVec* lcioClusters = new LCCollectionVec( LCIO::CLUSTER )  ;

  GenericHitVec<CalorimeterHit> h ;

  GenericClusterVec<CalorimeterHit> cl ;

  EnergyCut<CalorimeterHit> eCut( _eCut ) ;

  ZIndex<CalorimeterHit,100> zIndex( -4300. , 4300. ) ;

  NNDistance< CalorimeterHit, float> dist( _distCut )  ;

  LCIOCluster<CalorimeterHit> converter ;



  // create a vector of generic hits from the collection applying an energy cut
  for( StringVec::iterator it = _colNames.begin() ; it !=  _colNames.end() ; it++ ){

    LCCollection* col =  evt->getCollection( *it )  ;

//     addToGenericHitVec( h , col , eCut ) ;
     addToGenericHitVec( h , col , eCut , zIndex ) ;
  }

  // cluster the hits with a nearest neighbour condition
  cluster( h.begin() , h.end() , std::back_inserter( cl )  , &dist ) ;

  // create lcio::Clusters from the clustered GenericHits
  std::transform( cl.begin(), cl.end(), std::back_inserter( *lcioClusters ) , converter ) ;


  evt->addCollection( lcioClusters , _outputColName ) ;


  _nEvt ++ ;

  clock_t end =  clock () ;

  std::cout << "---  clustering time: " <<  double( end - start ) / double(CLOCKS_PER_SEC) << std::endl  ;
}
```

Summary

- coding guidelines are there to help you
- modular programming is an essential prerequisite for collaborative development
- when programming always stay focus on your problem at hand but keep your fellow colleagues in the back of your head:
  - how can this code be made a bit more abstract so that it can be used by the ILC community