RUN1C++ - moving to OO software environment

P.Murat, INFN-Pisa and ITEP-Moscow
April 4, 1996

1 Introduction

- It is quite likely that CDF software environment in Run II will be based on Object-Oriented (OO) language.
  The only question left now is

  to which extent ?

- It is very difficult to quantify the gain we get by switching to OO language.
  - industry experience
  - my personal experience: gain is extremely large.
  - the only motivation for changing the programming language :

    if using C++ allows to obtain physics results faster than with FORTRAN, than we should switch to it. Otherwise it is not necessary
  - However, the problem is that this can't be proved theoretically...
  - Nobody having experience with both OO and CO (Code-Oriented) languages claims that they are comparable...

2 how to proceed ?

- boundary conditions :
  - we are in the working experiment, we are doing physics analysis, we are working on the upgrade projects etc
  - the transition should be made adiabatically, without stopping any work in progress
- all the software, used by the collaboration could be divided into several structural layers:
  - layer, at which the the data model is being implemented (handling the data, I/O ...). For CDF it is provided by YBOS;
  - reconstruction layer;
  - analysis layer;

  of course, there is an overlap (sometimes - significant) between the different layers
- Analysis software (common for the collaboration):
  - input : reconstructed tracks, jets, vertices...
    output : reconstructed physical objects : electrons, muons, B-mesons ...
  - no other layer on top of it
  - used by almost everybody
  - relies heavily on the underlying layers (mostly - reconstruction)
  - easy to modify without any serious consequences
- reconstruction layer:
  - knows everything about the data representation
  - changes are being made only by a relatively small number of people;
  - any change can seriously impact all the analysis software (for example, change in the reconstruction bank format)
- data model layer:
  - modified only by a few experts
  - any change can screw up everything...

- therefore it is natural to start switching to OO from the analysis level moreover - it is the place where should finally get all the advantages, otherwise there is no need to do it at all
- nice feature of OO languages is that definitions and implementation are separated:

```
RECONSTRUCTED_TRACK track;

double pt = track.pt();
```

  it is totally out of interest at the analysis level how **track.pt()** looks inside, therefore if we can provide a C++-description of the reconstructed objects, used as an input at the analysis level, all the following changes will be transparent for the majority of thet collaboration.

- then we can replace the reconstruction level, by writing the C++-interfaces to the data model layer, but keeping it's implementation the moment
- and, finally we can replace FORTRAN code by C++ code at the software layer, implementing the data model

3 RUN1C++ package

- it is a set C++-written interfaces to the output of CDF reconstruction, which provides a "human language" description of the reconstructed objects and allows to perform a physics analysis **purely** in C++;
- it contains a huge number of interfaces to the different parts of CDF-written software
- it contains C++ interfaces to CERN software (HBOOK, ZEBRA, HIGZ...)
- it is totally integrated into CDF software environment (Analysis Control - A_C, modules) we learned write A_C modules in C++ and to link it with A_C
- the portability of mixed-language (FORTRAN/C/C++) code across the UNIX platforms has been checked : it works with under IRIX and AIX
- different debugging issues has been tested :
  - not any debugger supports simultaneously C/C++ and FORTRAN:
  - **gdb** , for example, on IRIX 5.3, allows to debug mixed-language executable, but doesn't recognize FORTRAN globals...
  - fortunately IRIX **cvd** does all we need ...;

```
/* -*- Mode: C++ -*-                                                          */
/*----------------------------------------------------------------------------*/
//  C/C++ interface to CERN PACKLIB routines                                  */
/*----------------------------------------------------------------------------*/

#ifndef _PACKLIB_H_
#define _PACKLIB_H_

#include <string.h>



#ifdef __cplusplus
extern "C" {
#endif

                    /**********************************************************/
                    /*      general purpose routines                          */
                    /**********************************************************/

  void   vfill_ (float *x, int* n, float* val);
  float  vmin_  (float *x, int *n);
  float  vmax_  (float *x, int *n);
  void   vzero_ (float *x, int *n);


#ifdef __cplusplus
};
#endif


#ifdef __cplusplus

/******************************************************************************/
/*     FORTRAN-like calling interfaces (C++ style)                            */
/******************************************************************************/

inline void  call_vfill(float& x, int n, float val) { vfill_ (&x,&n,&val); }
inline void  call_vzero(float& x, int n)            { vzero_ (&x,&n);      }

inline float vmin (float& x, int n) { return vmin_ (&x,&n);  }
inline float vmax (float& x, int n) { return vmax_ (&x,&n);  }

#endif  /* __cplusplus   */


#endif  /* _PACKLIB_H_  , end of packlib.h  */
```

4 Object number 1: PARTICLE

- From the very general point of view the main advantage of OO-approach to software development consists in that it allows to speak in terms of objects and operate with objects.
- one of the objects (probably, the first one!), which, I think, most of the HEP people want to be defined in terms of OO-language, is "particle". It seems that the definition of particle in any OO-language is one of the key checks of its applicability: if handling particles will not be more convenient, then in FORTRAN, we should forget about this language. That is why in this presentation I concentrate on handling particles.
- there are **MANY** different objects we call "particles", and it seems that any attempt to define just one, "basic", object will immediately fail.

  We talk, for example, about

  - particle, as it described in CDF particle database. (CDF has its own PDG-like scheme...): for this particle we have only PDG-like information defined;
  - generated particle (after MC generator - HERWIG/PYTHIA/...): this particle has its 4-momentum defined and "remembers" its "parents"
  - simulated particle (after simulation - QFL/CDFSIM): in addition to 4-momentum and history this particle has defined its production and decay vertices;
  - reconstructed particle (at the analysis stage): the definition strongly depends on the physical analysis considered...
- there is no one-to-one correspondence between these "particles", some of them have many in common, they may inherit from each other, refer to each other, but there is a lot of differences between them as well.
  This means, that for different purposes we need several different definitions of what the "particle" is, and there is nothing wrong in it.

4.1 CDFDB_PARTICLE description

```
class CDFDB_PARTICLE {

friend class CDF_PARTICLE_CLASS;
friend class CDF_PARTICLE_DATABASE;
//------------------------------------------------------------------------------
// data members
//------------------------------------------------------------------------------
 public :

  struct DECAY_CHANNEL_ {
    int    SelectionCode;
    int    Branching;                         // (branching ratio)*10**4
    int    NDaughters;                        // number of daughters
    int    DaughterCDFCode[MAX_DAUGHTERS];    // CDF codes of daughters
  };

  int      Number;                 // sequential number within the class
  int      CDF_Class;              // CDF particle class number
  int      CDF_Code;               // CDFCode = 100*CDFClass+Number
  char     Name[20];               // particle name
  int      AntiParticle;           // CDF code of antiparticle
  double   Mass;                   // particle mass in GeV
  double   Charge;                 // charge in units of abs electron ch
  double   Lifetime;               // proper lifetime in nanoseconds
  double   Width;                  // total width in GeV
  char     GenerationFlag;
  char     SimulationFlag;
  int      DecayLevel;
  int      Isospin;                // isospin*2 (times 2)
  int      I3;                     // i3*2
  int      Strangeness;            // flavours are defined if i3 != -1
  int      Charm;                  // 1 or -1 for C-particles, 0 for others
  int      Beauty;                 //
  int      Top;                    //
  int      Y;                      // X,Y: quantum numbers for 4-th generation
  int      X;                      //
  int      DecayChannels;          // number of decay channels
  DECAY_CHANNEL_  (*Decay)[10];    // pointer to the decay channels
//------------------------------------------------------------------------------
//  functions
//------------------------------------------------------------------------------
public:
  double charge() { return Charge; };  // just an example
}
```

4.2 GENP class description

```
struct GENP_PARTICLE_{
  int    CDFCode;                       // CDF particle code
  int    MotherCode;                    // coded address of mother particle
  int    DaughterCode;                  //
  int    GenerationCode;
  int    DecayCode;
  float  Px;                            // these are float because it is
  float  Py;                            // pointer to GENP bank...
  float  Pz;
  float  Mass;
};

class GENP {
friend class GENP_PARTICLE;
//------------------------------------------------------------------------------
// data members:
//------------------------------------------------------------------------------
private:
  YBOS_BANK* Bank;
  int        Initialized;

public:
                                        // total number of generated particles,
                                        // including  quarks, gluons, etc.
  int     NEntries;
                                        // number of physical particles
                                        // in the final state
  int     NParticles;
                                        // pointer to array of particle descriptions
  GENP_PARTICLE_  (*GenpData)[1000];
                                        // (*Number)[i]: sequential number of the
                                        // i-th final state particle in the entry
                                        // table,
                                        // (*Particle)[(*Number)[i]] is the pointer
                                        // in GENP to 'i'-th final state particle
  int        (*Number)[1000];
//------------------------------------------------------------------------------
// methods:
//------------------------------------------------------------------------------
      .............
};
```

4.3 GENP_PARTICLE class description

```
class GENP_PARTICLE : public LORENTZ_VECTOR {
//-----------------------------------------------------------------------------
//  data members:
//-----------------------------------------------------------------------------
private:

  GENP_PARTICLE_*    Gp;                  // pointer to the particle data in GENP
  CDFDB_PARTICLE*    CdfdbParticle;       // pointer to the particle description
                                          // in the CDF particle database
  double             Boost;               // E/m

  double             Code;                //
  int                FirstDaughter;       // 1st  daughter # (in  GENP)
  int                LastDaughter;        // last daughter # (in  GENP)
  int                JetNumber;           // pointer to the the closest jet
  int                Iobsp;               // # of particle in OBSP
//-----------------------------------------------------------------------------
//  methods:
//-----------------------------------------------------------------------------
public:
                                          // constructors
  GENP_PARTICLE(int i);
  GENP_PARTICLE(GENP_PARTICLE_* p);

  double Charge   () { return CdfdbParticle->Charge; };

  double mass     () { return (double) Gp->Mass; };

  int IsElectron() {
    return (( Gp->CDFCode == CDF_ELECTRON) || (Gp->CDFCode == CDF_POSITRON));
  };

  int IsWboson() {
    return ( (Gp->CDFCode == CDF_W_PLUS) || (Gp->CDFCode == CDF_W_MINUS));
  };
};
```

4.4 MC class description

```
 class MC {

public:
  struct PARTICLE_{
    float  Px;                   // these are float because it is
    float  Py;                   // pointer to OBSP bank...
    float  Pz;
    float  Mass;
    float  Path;                 // particle path in the detector
    int    CDFCode;              // CDF particle code
    int    OrignVertexNumber;    // OBSV # of particle production vertex
    int    DecayVertexNumber;    // OBSV # of particle decay vertex
    int    GenpNumber;           // GENP # of the particle
    int    DecayCode;            // particle decay code
  };

  struct VERTEX_ {
    float  X;
    float  Y;
    float  Z;
    float  Time;
    int    Parent;               // parent particle OBSP number
    int    FirstDaughter;        // OBSP number of the 1st daughter
    int    LastDaughter;         // OBSP number of the last daughter
  };

private:

  YBOS_BANK*   _OBSP;            // pointer to OBSP bank
  YBOS_BANK*   _OBSV;            // pointer to OBSV bank

                                 // pointer to array of particle descriptions

  PARTICLE_   (*Particle)[MAX_MC_PARTICLES];

                                 // pointer to array of vertex descriptions

  VERTEX_     (*Vertex)[MAX_MC_VERTICES];
//--------------------------------------------------------------------------------
//  methods:
//--------------------------------------------------------------------------------
public:
             ..................
};
```

4.5 MC_PARTICLE class description

```
class MC_PARTICLE : public LORENTZ_VECTOR {

  MC::PARTICLE_*      Obsp;             // pointer to the particle data in OBSP
  MC::VERTEX_*        Origin;           // pointer to the patricle origin
  MC::VERTEX_*        DecayPoint;       // pointer to the particle decay point
  double              Boost;            // particle E/m
  CDFDB_PARTICLE*     CDFDB_Particle;   // pointer to CDF particle database
                                        // description of this particle
  double              Code;             //

public:

  MC_PARTICLE(int i);
                                        // point where particle "was born"
                                        // (the decay point of its 'mother')

  V3      origin() { return V3(Origin->X,Origin->Y,Origin->Z); };

                                        // decay point of the particle

  V3      decayPoint() {return V3(DecayPoint->X,DecayPoint->Y,DecayPoint->Z); };

                                        // particle path in the detector

  double path      () { return (double) Obsp->Path; };

  double & code    () { return Code; };

  int     CDFCode   () { return Obsp->CDFCode  ;           };
  int     DecayCode () { return Obsp->DecayCode;           };
  double  Charge    () { return CDFDB_Particle->Charge;    };
  int     CDF_Class () { return CDFDB_Particle->CDF_Class; };
  double  Lifetime  () { return CDFDB_Particle->Lifetime ; };

  double mass() { return (double) Obsp->Mass; };
};
```

5 Examples

- MC particle lifetime :

```
MC_PARTICLE p(i);
double time   = p.Lifetime();
```

- cos(theta) between 3-momenta of two particles:

```
GENP_PARTICLE p1(i);
GENP_PARTICLE p2(j);
double cos_theta = p1.v3().ort()*p2.v3().ort();
```

- effective mass of 2 particles:

```
GENP_PARTICLE p1(i);
GENP_PARTICLE p2(j);

double m12       = (p1.v4()+p2.v4()).mass();
```

- Selecting the high-Pt W's ($P_t > 20$GeV) in GENP bank:

```
for (int i=0; i<Genp.NParticles; i++) {
  GENP_PARTICLE p(i);
  if (p.IsWboson() && p.pt() > 20.) {
                                           // this is W-boson with
                                           // Pt > 20 GeV
         ..........

    };
  };
};                                         // 'p' is deleted automatically
                                           // in the end of loop
```

- fill column-wise ntuple for high-Pt jets (from B-tagging module) :

```
class JET : public LORENTZ_VECTOR {
   ...............
};
class JET_LIST {
};
  .............

  JET_LIST JetList;
  ................
  struct JET_NTUPLE_RECORD_{
    float eta;                          // jet eta
    float phi;                          // jet phi
    float et;                           // jet Et
    float svxt;                         // number of SVX tracks
    float vz;                           // z-coord of primary vertex
    float path;                         // Lxy(B) for Bjet
  };
  JET_NTUPLE_RECORD_ Jt;
  ............
  for (int i=0; i<JetList.NhighPtJets; i++) { // loop over jets
    JET& jet  = *JetList.HighPtJet(i);
    Jt.eta    = jet.eta();
    Jt.phi    = jet.phi();
    Jt.et     = jet.et();
    Jt.svxt   = jet.svxTracks();
    Jt.vz     = VxPrim.z();
                                        // *** B-meson Lxy (for MC Bjet
    GENP_PARTICLE* b = jet.Particle;
    if (b)
      Jt.path = b->decayLength()*b->pt()/b->v3().mod();
    else
      Jt.path = UNDEFINED;

     call_hfnt(2);           // interface to FORTRAN ``call hfnt(2)''
  };
```

6 The most important advantages of C++ over FORTRAN observed

- C++ is much more "human" language - this has a lot of consequences:
- first of all, the natural natural description of the data translates into much more clear structure of the code itself. Of course, it is quite possible to write FORTRAN code in well-structured manner, and in fact, in this case it will be very close to code, written in C++. But it requires a very high programming culture, very high discipline and a lot of efforts on organizing the data, while with C++ it comes up naturally. Using C++, it's rather difficult to write the code in non-structured manner - language itself implicitly dictates structured approach (just try and you immediately feel it - it's a very strange feeling of that your code is badly organized!)
- errors: I beleve, that most errors, done by so-called "experts", are due to the loss of attention, to that the programmer has forgotten something, and other things like this.
  Writing code in C++ requires much less things to be remembered!
  In many cases the language (via compiler) prevents a programmer from making an error (typical examples: function prototyping, protected members, function members used to access the object data)
- symbolic debugging : debugger shows the objects with references therein - a lot of necessary variables is shown automatically - you understand, how important it is!. (IRIX cvd)
  Moreover, for some strange reason C/C++ code is analized by any debugger, I've tryed, much more fast, than FORTRAN code;

- what I especially enjoy, are the operations with 4-vectors and all the linear algebra stuff:

```
MATRIX   matrix;
VECTOR   v1;
VECTOR   v2;
VECTOR   v3;

c   = m*(a+b);
```

It is a great advantage over FORTRAN;

7 Potential problems

- the necessity to write destructors. If a constructor for some class dynamically allocates memory for one of the data members, it's necessary to write the coresponding destructor to deallocate this memory. This is the source of potential errors, which often are very difficult to localize. Compiler doesn't help here and one should be very careful;
- absense of many "standard" libraries; non-convenience of some classes, written by others (many people are just learning, how to do it!);
- for FORTRAN-minded person it's pretty easy to write VERY inefficient code - just pass parameters by value! But this is very easy to learn...;
- - debugging 3-rd party software:
  CLHEP : STACK class under linear algebra class ...
  issue of efficiency.

8 Conclusions

- How to understand whether the tool you are using is helpful or not? Probably, one of the best checks is to try not to use it and to see, how do you feel yourself without it.
  It comes from my personal experience, that in many cases I feel very uncomfortable, when I need to code something in pure FORTRAN. But it is not due to I don't know the language - I just know, that the same piece of code may be done much more easily and elegantly in C++ and my internal feeling is protesting...
- for any HEP physicist, who knows C++, I guess, there is no question at all : C++ suits better for our needs, than any code-oriented language (like FORTRAN, C etc).
- the best test laboratory for the new language : software for Run II upgrades
- "heart" of CDF - its tracker
  already now we have track reconstruction software for RunII Intermediate Fiber Tracker (IFT) which is written totally in C++;
- IFT reconstruction has been recently integrated with the SVXII reconstruction code, written in FORTRAN (essentially based on RUN1C++ interfaces)
- there are other C++ on-going software projects :
  - C++ standalone reconstruction for SVXII (J.Bougreau)
  - an attempt to rewrite YBOS in C++ (R.Kennedy)
  - there is a C++ effort in Univ of Penn CDF group