Automatic Schema Evolution in ROOT

CHEP09: Prague, 24 March 2009

Philippe Canal/FNAL

René BRUN/CERN, Lukasz Janyst/ CERN,
Jérôme Lauret/BNL, Valeri Fine/BNL

BROOKHAVEN NATIONAL LABORATORY

CERN

Apples And Oranges

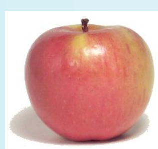
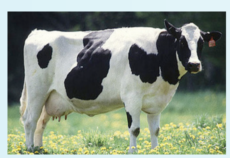
Simple Automatic Schema Evolution.

- Easily lets you transform  into 

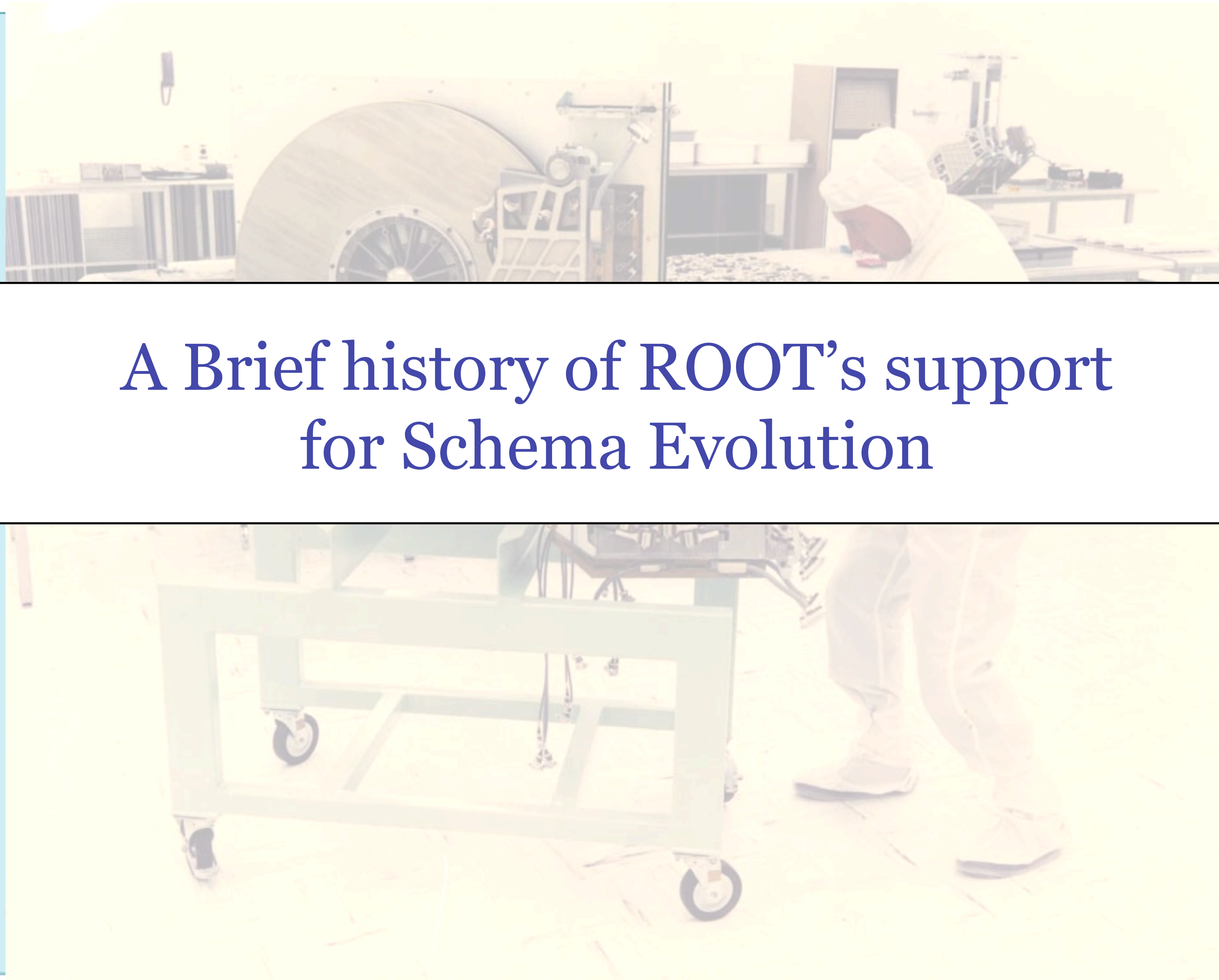
Hand Coded Schema Evolution

- Allows to transform  into 
- Requires specific coding for each type of apple and orange.

Complex Automatic Schema Evolution

- Allow almost any kind of transformation
- even  to 

A Brief history of ROOT's support for Schema Evolution



ROOT I/O History

1995

Version 0.9

- Hand-written Streamers

1996

Version 1

- Streamers generated via rootcint
- Support for Class Versions

1998

Version 2.25

- Support for ByteCount
- Several attempts to introduce automatic class evolution
- Simple support for STL
- Only hand coded and generated streamer function, Schema evolution done by hand
- I/O requires : ClassDef, ClassImp and CINT Dictionary

2000

2001

Version 2.26 – 3.00

- **Automatic schema evolution**
- **Use TStreamerInfo (with info from dictionary) to drive a general I/O routine.**
- **Self describing files**
- **MakeProject** can regenerate the file's classes layout

ROOT I/O History

'2002

Version 3.03/05
- **Lift need for** ClassDef and ClassImp for classes not inheriting from TObject
- **Any non TObject class** can be saved inside a TTree or as part of a TObject-class
- **TRef/TRefArray**

'2004

Version 4.00/08
- Automatic versioning of 'Foreign' classes
- Non TObject classes can be saved directly in TDirectory

Version 4.04/02
- Large TTrees, TRef autoload
- TTree interface improvements, Double32 enhancements

'2005

Version 5.08/00
- Fast TTree merging, Indexing of TChains, **Complete STL support.**

'2006

Version 5.12/00
- Prefetching, TTreeCache
- TRef autoderefencing

'2007

Version 5.16/00
- Improved modularity (libRio)

'2008

Version 5.22/00
- **Data Model Evolution** (brought to you courtesy by BNL/STAR/ATLAS)

Early Days

At first, streamers needed to be fully written by hand

- Very labor intensive and error prone.

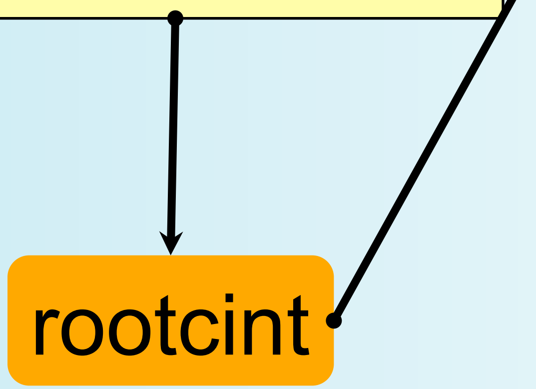
Dictionaries became the corner-stone of the I/O

- Allowed streaming of user class with minimal intrusion and no complex ddl system.
- rootcint generated default C++ Streamer function

- But all schema evolution required to maintain the streamer functions by hand

Streamers in 0.90/08

```
class TAxis :
   public TNamed,
   public TAttAxis
{
private:
  Int_t        fNbins;
  Axis_t       fXmin;
  Axis_t       fXmax;
  TArrayF      fXbins;
  …
  ClassDef(TAxis,1);
};
```




```
void TAxis::Streamer(TBuffer &b)
{
   if (b.IsReading()) {
      Version_t R__v = b.ReadVersion();
      TNamed::Streamer(b);
      TAttAxis::Streamer(b);
      b >> fNbins;
      b >> fXmin;
      b >> fXmax;
      fXbins.Streamer(b);
   } else {
      b.WriteVersion(TAxis::IsA());
      TNamed::Streamer(b);
      TAttAxis::Streamer(b);
      b << fNbins;
      b << fXmin;
      b << fXmax;
      fXbins.Streamer(b);
   }
}
```

Streamers in 0.90/08

```
class TAxis :
  public TNamed,
  public TAttAxis
{
private:
  Int_t        fNbins;
  Axis_t       fXmin;
  Axis_t       fXmax;
  TArrayF      fXbins;

  Int_t        fFirst;
  Int_t        fLast;
  …
  ClassDef(TAxis,2);
};
// New member fFirst and fLast.
```

```
void TAxis::Streamer(TBuffer &b)
{
   if (b.IsReading()) {
      Version_t R__v = b.ReadVersion();
      TNamed::Streamer(b);
      TAttAxis::Streamer(b);
      b >> fNbins;
      b >> fXmin;
      b >> fXmax;
      fXbins.Streamer(b);
      if (R__v > 3) {
         R__b >> fFirst;
         R__b >> fLast;
      }
   } else {
      b.WriteVersion(TAxis::IsA());
      TNamed::Streamer(b);
      TAttAxis::Streamer(b);
      b << fNbins;
      b << fXmin;
      b << fXmax;
      fXbins.Streamer(b);
   }
}
```

developer

Streamers in 2.25

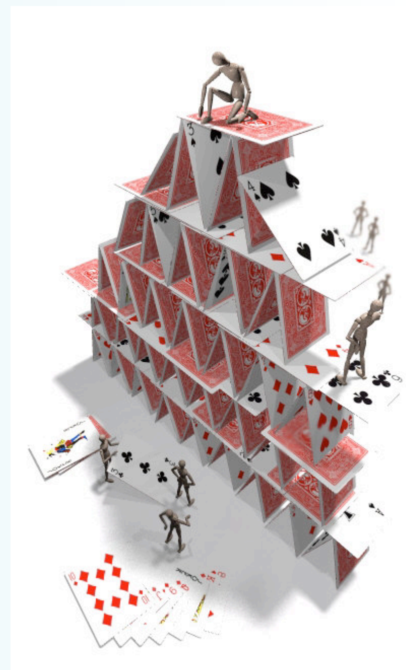
As of version 2.25 (1997), the ROOT streamers fully supports complex schema evolution.

However:

- They were becoming *overly complex* due to the increasing number of versions to be kept track of.
- They were not supporting forward compatibility

  There was no way to read in an older version of ROOT a file written with a newer version of ROOT.

- They needed to be updated for almost any small change in the classes.



Reading the object required access to the original compiled code.

2000 - StreamerInfo

ROOT Files are now self describing

- Dictionary for persistent classes written to the file when closing the file.
- ROOT files can be read by foreign readers (JAS for example)
- Support for Backward and Forward compatibility
- Files created in 2003 can be read in 2015
- Classes (data objects) for all objects in a file can be regenerated via ***TFile::MakeProject***
- Data can be read without the original code

Support for simple automatic schema evolution:

- Change the order of the members
- Change simple data type (float to int)
- Add or remove data members, base classes
- Migrate a member to base class

Streamers in 3.00 - StreamerInfo

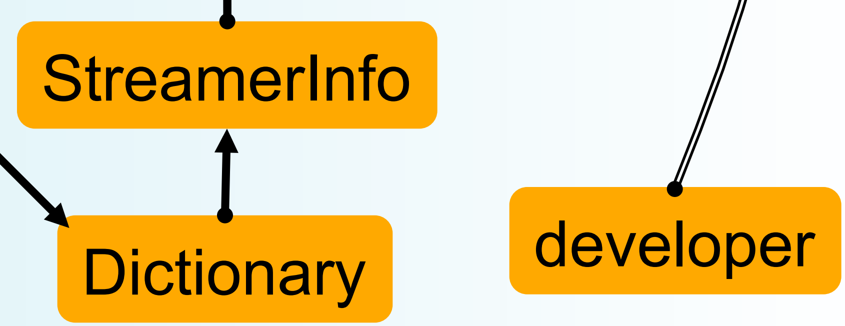
```
class TAxis :
   public TNamed,
   public TAttAxis
{
private:
   Int_t          fNbins;
   Axis_t         fXmin;
   Axis_t         fXmax;
   TArrayF        fXbins;
   Int_t          fFirst;
   Int_t          fLast;
   TString        fTimeFormat;
   Bool_t         fTimeDisplay;
   TObject       *fParent; //!
   …
   ClassDef(Taxis,7);
};
```

```
void TAxis::Streamer(TBuffer &R__b)
{
   // Stream an object of class TAxis.

   if (R__b.IsReading()) {
      UInt_t R__s, R__c;
      Version_t R__v = R__b.ReadVersion(&R__s, &R__c);
      fParent = 0;
      if (R__v > 5) {
         TAxis::Class()->ReadBuffer(R__b, this, R__v, R__s, R__c);
         return;
      }
      //====process old versions before automatic schema evolution
      ....
      //====end of old versions

   } else {
      TAxis::Class()->WriteBuffer(R__b,this);
   }
}
```

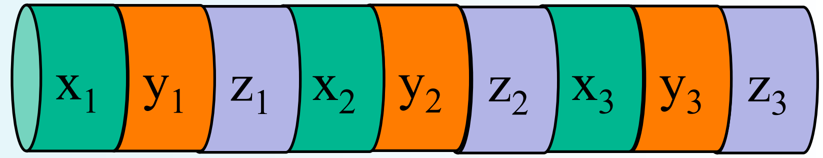
- Routine class maintenance does not require manual updates.
- Allow for pre and post streaming operation (setting a transient member)



Objectwise vs. Memberwise

Object wise Streaming:

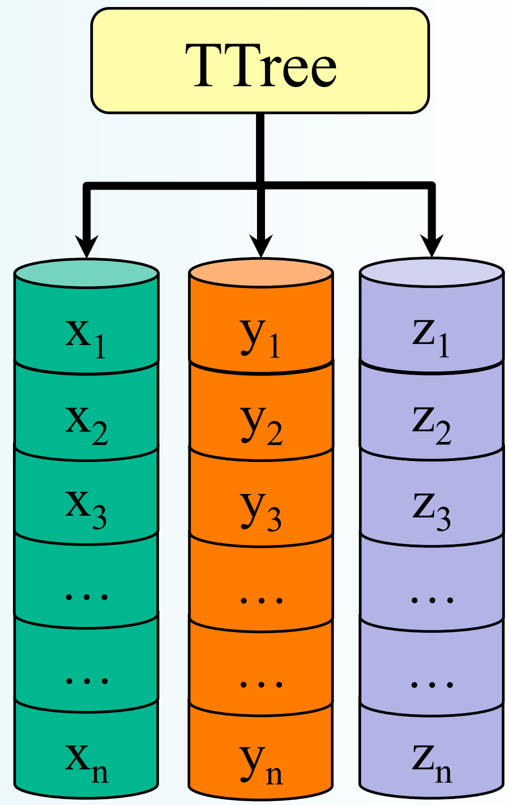
- For each object all data members are streamed sequentially in the same buffer.
- This is the original technique using Streamer functions.



Member wise streaming:

- For each member the value of this member for all objects is stored
- Each member has its own buffer
- Requires use of StreamerInfo
- Advantages:
  - Better compression
  - Better read/write time
  - Ability to read partial objects

Essential For Fast Analysis



Simple Automatic Schema Evolution

Support

- Changing the order of the members
- Changing simple data type (float to int)
- Adding or removing data members, base classes
- Migrating a member to base class

Limitations

- Handle only removal, addition of members and change in simple type
- Does not support complex change in type, change in semantic (like units)
- Further customization requires using a Streamer function
  - Allow complete flexibility including setting transient members

However they can ***NOT*** be used for member-wise streaming (TTrees)

Complex Automatic Schema Evolution

Complex Automatic Schema Evolution

Complex Automatic Schema Evolution solves existing limitations

- Assign values to transient data members
- Rename classes
- Rename data members
- Change the shape of the data structures or convert one class structure to another
- Change the meaning of data members
- Ability to access the ***TBuffer*** directly when needed
- Ensure that the objects in collections are handled in the same way as the ones stored separately
- Transform data before writing

☞ ***Make things operational also in bare ROOT mode***

☞ ***Supported in object-wise, member-wise and split modes.***

Complex Automatic Schema Evolution

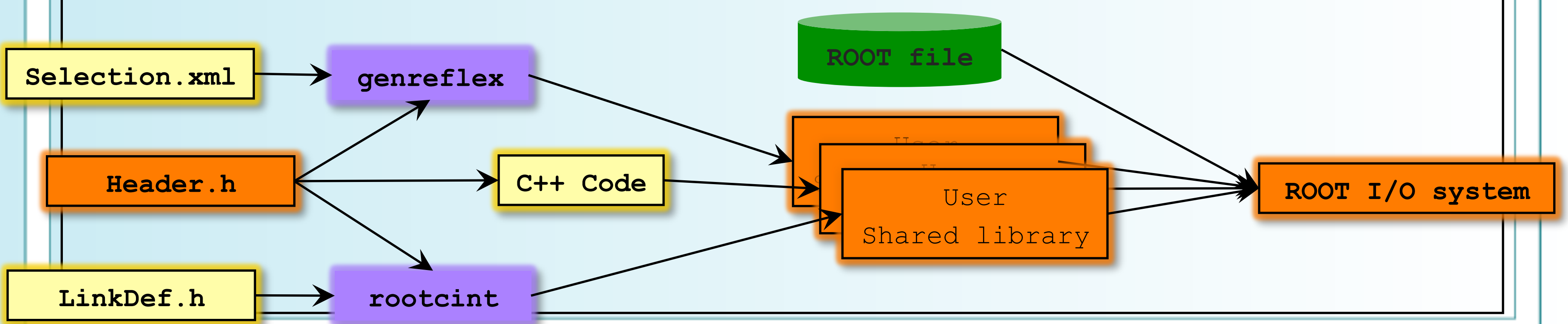
User can now supply a function to convert individual data members from disk to memory and rule defining when to apply the rules

A schema evolution rule is composed of:

- ***sourceClass***; version, checksum: identifier of the on disk class
- ***targetClass***: name of the class in memory
- ***source***: list of type and name of the on disk data member needed for the rule
- ***target:*** list of in memory data member modified by the rules.
- ***include***: list header files needed to compile the conversion function
- ***code***: function or code snippet to be executed for the rule

Rules can be registered via:

- LinkDef.h, Selection.xml, C++ API (via TClass), ROOT files



Dictionary Generation Syntax

Example of registering a rule from a LinkDef file:

```
#pragma read sourceClass="oldname" version="[1-]" checksum="[12345,23456]" \
   source="type1 var; type2 var2;" \
   targetClass="newname" target="var3" \
   include="<cmath> <myhelper>" \
   code="{ … 'code calculating var3 from var1 and var2' … }"
```

Example of registering a rule from a Selection.xml file:

```
<read sourceClass="oldname" version="[4-5,7,9,12-]" checksum="[12345,123456]"
      source="type1 var; type2 var2;"
      targetClass="newname" target="var3"
      include="<cmath> <myhelper>"
<![CDATA[
   … 'code calculating var3 from var1 and var2' …
]]>
</read>
```

C++ Syntax

Example of registering a rule from C++:

```
// Create the rule
rule = new TSchemaRule();
rule->SetSourceClass("oldname"); // Name of the class on file
rule->SetVersion("[1-");          // Set of version numbers this rule applies to
rule->SetChecksum("[12345]");     // Set of checkums this rules applies to
rule->SetSource("type1 var; type2 var2;"); // Where to get the info from
rule->SetTarget("var3");          // Name of the variable to set
rule->SetInclude("<cmath> <myhelper>");  // When needed to compile the code
rule->SetCode("{ … 'code calculating var3 from var1 and var2' … }");
rule->SetRuleType( TSchemaRule::kReadRule );
rule->SetReadFunctionPointer( functionptr ); // Alternative to the 'string' code.

// Register the rule
TClass::GetClass(newname)->GetSchemaRules(kTRUE)->AddRule(rule);
```

Setting A Transient Member

MyClass.h

```
class MyClass {
private:
  Type fComplexData;
  Double_t fValue; //! Calculated from fComplexData
  Bool_t fCached;  //! True if fValue has been calculated
public:
  double GetValue() { if (!fCached) { fValue = … ; }; return fValue; }
```

MyClassLinkDef.h

```
#pragma read sourceClass="MyClass" version="[1-]” source=””
   targetClass=”MyClass" \
   target=”fCached" \
   code="{ fCached = false; }"
```

This example shows how to initialize a transient member

`source=””` indicates that no input is needed

`version="[1-]` indicates that the rule applies to all versions of the class

`target=”fCached”` indicates which member will be modified by the rule

This resolves the outstanding issues where transient members are currently not updated when (re-)reading an object from a split branch

Merging Several Data Members

MyClass.h

```
class MyClass {
private:
  int fX;
  int fY; // Values between 0 and 999
  int fZ; // Values between 0 and 9
public:
  int GetX() { return fX; }
  int GetY() { return fY; }
  ClassDef(MyClass,8);
}
```

MyClass.h

```
class MyClass {
private:
  long fValues; // Merging of fX, fY and fZ
public:
  int GetX() { return fValues / 1000; }
  int GetY() { return (fValues%1000)-GetZ(); }
  int GetZ() { return fValues % 10;
  ClassDef(MyClass,9);
}
```

MyClassLinkDef.h

```
#pragma read sourceClass="MyClass" version="[8]" targetClass="MyClass " \
   source="int fX; int fY; int fZ" target="fValues" \
   code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```

In ***MyClass*** version 9, to save memory space, 3 data members were *merged*.

`source="int fX; … "` indicates the types and name of the original members.

`onfile.fX` gives access to the value of ***fX*** read from the buffer.

Renaming A Class

MyClass.h

```
class MyClass {
private:
  int fX;
  int fY; // Values between 0 and 999
  int fZ; // Values between 0 and 9
public:
  int GetX() { return fX; }
  int GetY() { return fY; }
  ClassDef(MyClass,8);
}
```

Properties.h

```
class Properties {
private:
  long fValues; // Merging of fX, fY and fZ
public:
  int GetX() { return fValues / 1000; }
  int GetY() { return (fValues%1000)-GetZ(); }
  int GetZ() { return fValues % 10;
  ClassDef(Properties,2);
}
```

PropertiesLinkDef.h

```
#pragma read sourceClass="MyClass" version="[9]" targetClass="Properties"
#pragma read sourceClass="MyClass" version="[8]" targetClass="Properties" \
  source="int fX; int fY; int fZ" target="fValues" \
  code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```

To clarify its purpose the class needed to be renamed.

- ***sourceClass*** and ***targetClass*** are respectively ***MyClass*** and ***Properties***
- 1st rule indicates that version 9 of ***MyClass*** can be read directly into a ***Properties*** object using only the simple automatic schema evolution rules.
- 2nd rule indicates that in addition to the simple rules, a complex conversion needs to be applied when reading version 8 of ***MyClass*** into a ***Properties*** object.

Complex Evolution – Nested Objects

The same *version* of a containing class can hold several *versions* of the nested object's class.

- ***Event*** version 2 contains an extended ***Track***
  - The ***Track*** class underwent a couple of updates while ***Event*** did not change
- ***Event*** version 3 contains
  - ***fCompactTrack***– a more compact ***Track***
  - ***fId*** – with information that used to be kept in the extended ***Track***

```
#pragma read sourceClass="Event" version="[2]" targetClass="Event" \
   source="Track fTrack;" target="fId; fCompactTrack;" \
   code="{ if( onfile.fTrack->GetVersion() == 3 ) \
           { \
              fId = onfile.fTrack->GetMember<double>( id_fTrack_fB) + \
                    onfile.fTrack->GetMember<double>( id_fTrack_fC ); \
              onfile.fTrack->Load( fCompactTrack ); \
           } \
           else if ( onfile.fTrack->GetVersion() == 4 ) \
           { \
              fId = onfile.fTrack->GetMember<double>( id_fTrack_fB); \
              onfile.fTrack->Load( fCompactTrack ); \
           }; }"
```

Copy data from Track to fCompactTrack by applying all the registered rules to evolve from Track to CompactTrack

Analysis Backward & Forward Compatibility

Time ***T1*:**

- ***MyClass*** has fPx, fPy, fPz
- write file *t1.root*
- write analysis *work1.C* using fPx, fPy, fPz

Time ***T2*:**

- ***MyClass*** has fR, fT, fP
- write file *t2.root*
- write analysis *work2.C* using fR, fT, fP

Backward Compatibility:

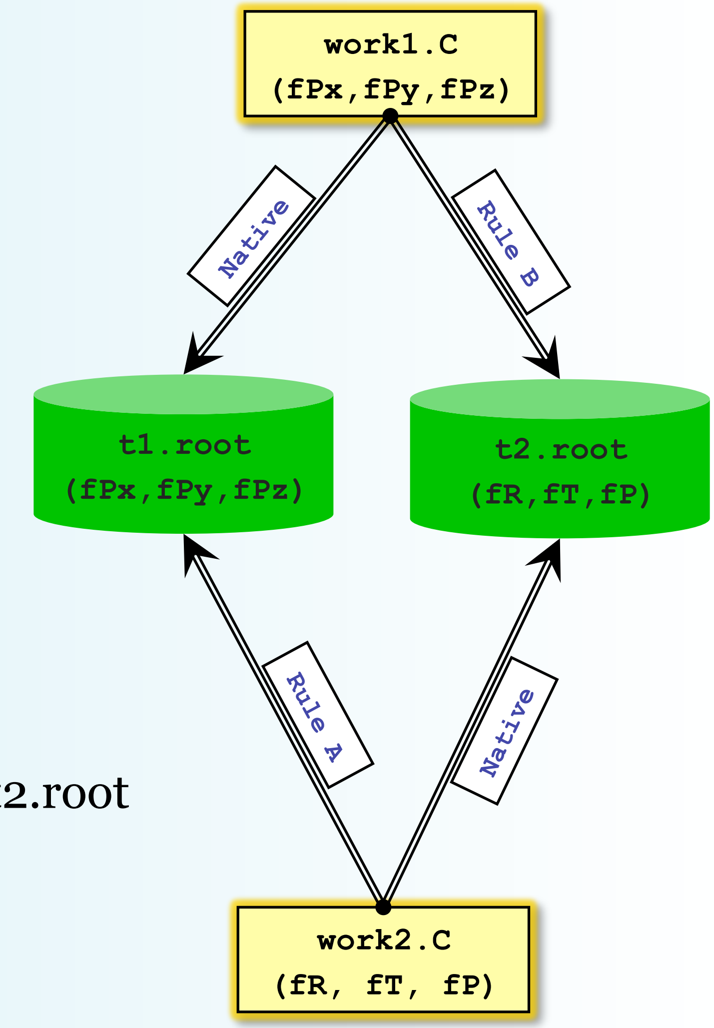
```
Rule A: (fPx, fPy,fPz) -> (fR, fT, fP)
```

- The user can run work2.C on **t1.root** or t2.root

Forward Compatibility:

```
Rule B: (fR, fT, fP) -> (fPx, fPy,fPz)
```

- The user can run work1.C on t1.root or **t2.root**



Summary

- New Complex Schema Evolution:
  - ✓ Increase flexibility and performance when reading old files.
  - ✓ Gives possibility to perform complex evolution even without user classes, the information being in the ROOT file
  - ✓ Powerful
  - ✓ Fun