European School of Instrumentation in Particle & Astroparticle Physics

European Scientific Institute

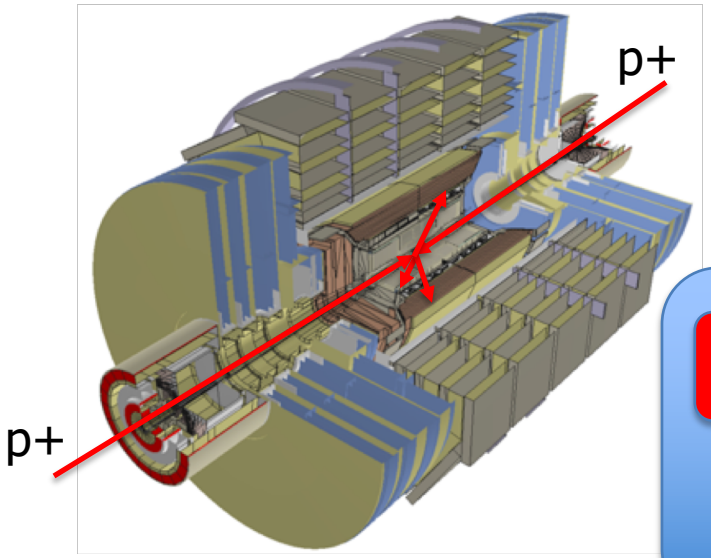
Detector Simulation

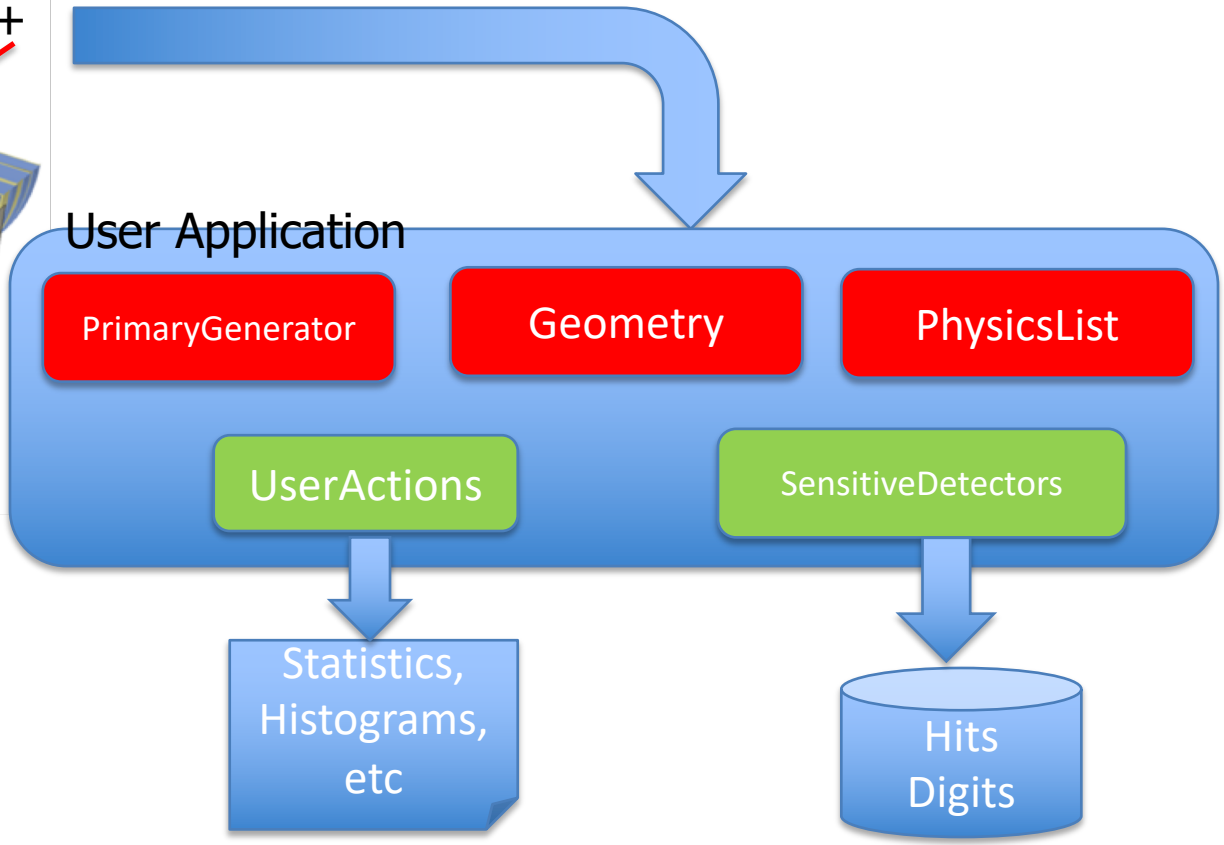
User Actions, Hits and Digits

Witek Pokorski
Alberto Ribon
CERN

11-12.02.2019

What do we need to run simulation?

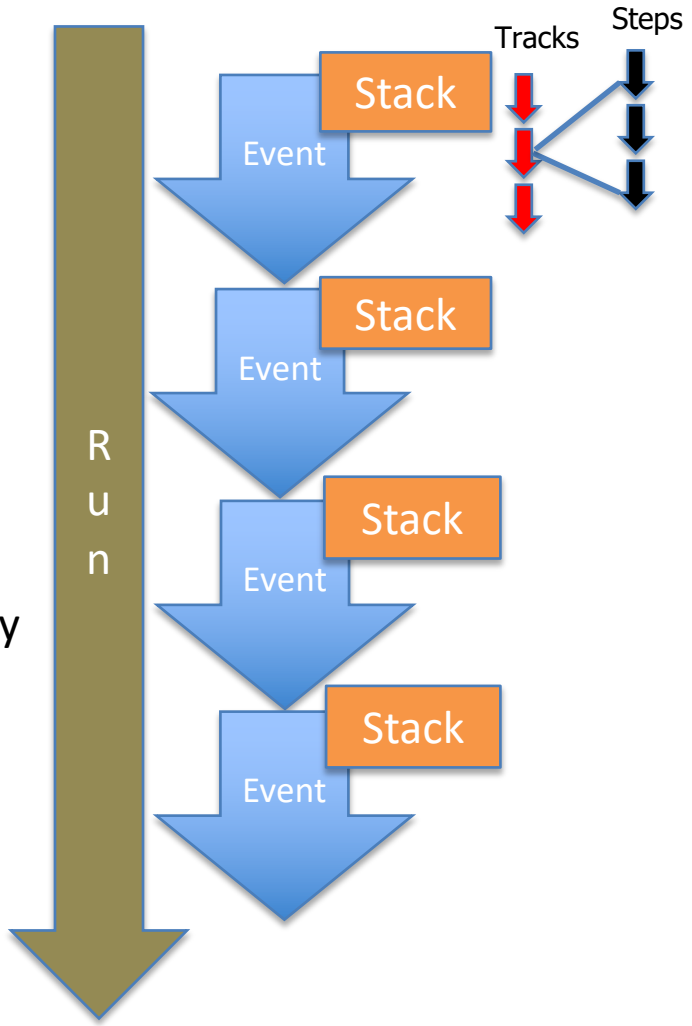




- Given geometry, physics and primary track generation, Geant4 does proper physics simulation "silently".
  - You have to add a bit of code to extract information useful to you.
- The user action classes, if provided, are called by Geant4 kernel during all phases of tracking

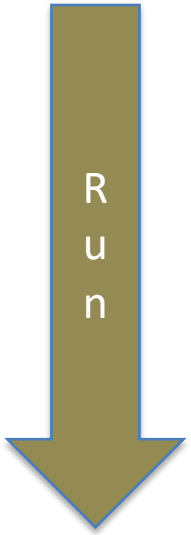
User Actions - Overview

- mandatory Users actions classes
  - G4VUserActionInitialization
  - G4VUserPrimaryGeneratorAction
- optional Geant4 User Action classes
  - G4UserRunAction
  - G4UserEventAction
  - G4UserTrackingAction
  - G4UserSteppingAction
  - G4UserStackingAction
- fully customizable (empty by default)
- the user action classes are used to setup and/or modify the simulation or collect information about the run
  - allow to take actions specific for the given simulation
    - simulated only relevant particles
    - save specific information, fill histograms
    - speed-up simulation by applying different limits

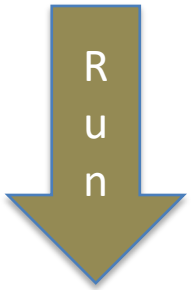


G4UserRunAction (1/2)

- virtual G4Run* GenerateRun()
  - This method is invoked at the beginning of BeamOn.
  - User hook to provide derived G4Run and create his/her own concrete class to store some information about the run
  - Ideal place to set variables which affect the physics table (such as production thresholds) for a particular run, because GenerateRun() is invoked before the calculation of the physics table.
- virtual void BeginOfRunAction(const G4Run*)
  - Invoked before entering the event loop
  - Typical use of this method would be to initialize and/or book histograms for a particular run
  - This method is invoked after the calculation of the physics tables



G4UserRunAction (2/2)



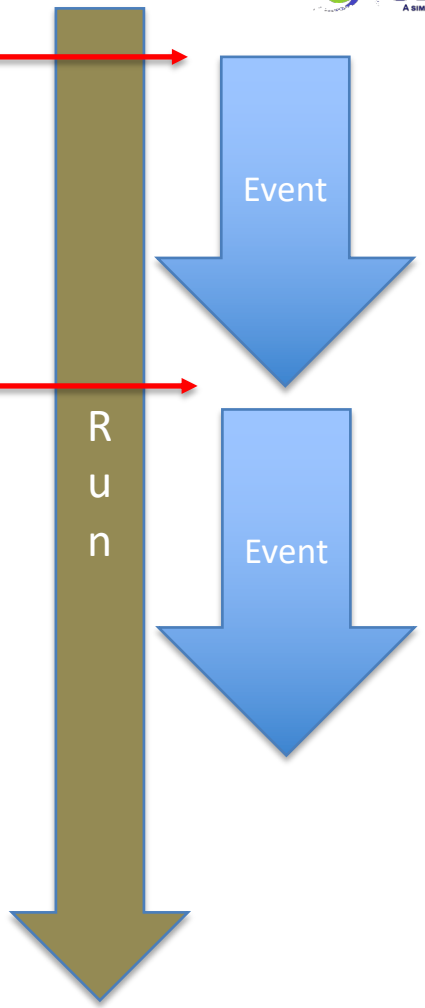
- virtual void EndOfRunAction(const G4Run*)
  - This method is invoked at the very end of the run processing
  - It is typically used for a simple analysis of the processed run
- virtual void SetMaster(G4bool val=true)
- G4bool IsMaster()
  - Commonly, a MT simulation will have a master-thread instance and a worker thread instance — provides ability to discern whether instance is for worker or master thread

G4UserEventAction

- virtual void BeginOfEventAction(const G4Event*)
  - This method is invoked before converting the primary particles to G4Track objects
  - A typical use of this method would be to initialize and/or book histograms for a particular event
- virtual void EndOfEventAction(const G4Event*)
  - This method is invoked at the very end of event processing
  - Typically used for a simple analysis of the processed event
  - If the user wants to keep the currently processing event until the end of the current run, the user can invoke

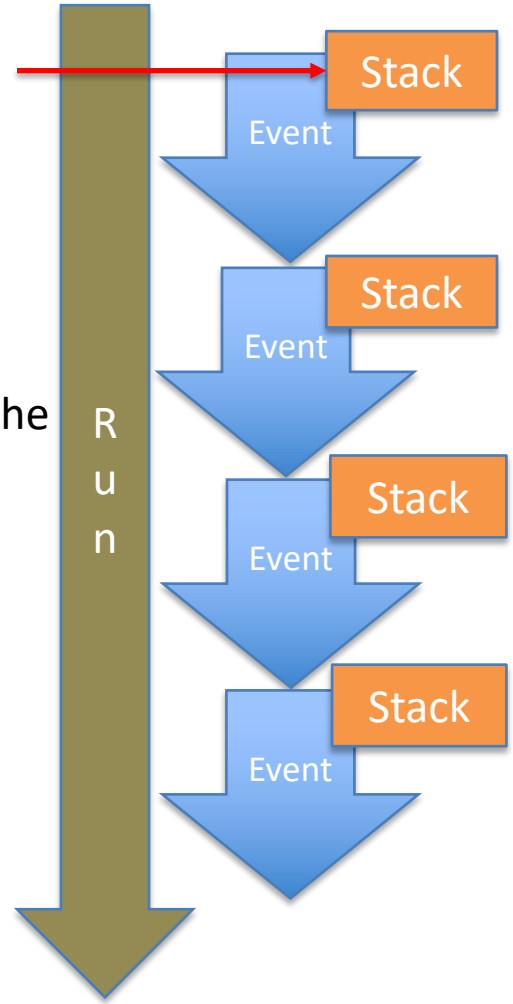
    G4EventManager::GetEventManager()->KeepTheCurrentEvent()

    so that it is kept in G4Run object.
    - can be used for visualization of particular events



G4UserStackingAction (1/2)

- G4UserStackingAction is a user-hook to reorder the priority of the particle stack
- virtual G4ClassificationOfNewTrack ClassifyNewTrack(const G4Track*)
  - invoked by G4StackManager whenever a new G4Track object is ”pushed” onto a stack by G4EventManager
  - Returns an enumerator whose value indicates to which stack the track should be sent. Value is determined by the user from four possible values
  - fUrgent — track is placed in urgent stack
  - fWaiting — track is placed in the waiting stack (until urgent is empty)
  - fPostpone — track is postponed to next event
  - fKill — track is deleted immediately and not stored

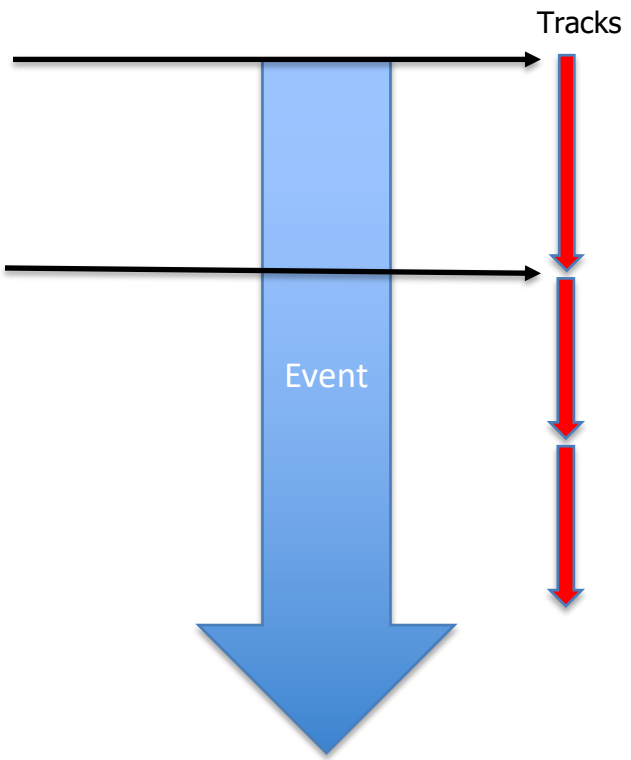


G4UserStackingAction (2/2)

- virtual void NewStage()
  - Invoked when the urgent stack is empty and the waiting stack contains at least one G4Track object
  - User may kill or re-assign to different stacks all the tracks in the waiting stack [G4StackManager::ReClassify()]
  - If no user action is taken, all tracks in the waiting stack are transferred to the urgent stack
  - The user may decide to abort the current event here
- virtual void PrepareNewEvent()
  - Invoked at the beginning of each event
  - At this point no primary particles have been converted to tracks, so the urgent and waiting stacks are empty
  - However, there may be tracks in the postponed-to-next-event stack; for each of these the ClassifyNewTrack() method is called and the track is assigned to the appropriate stack

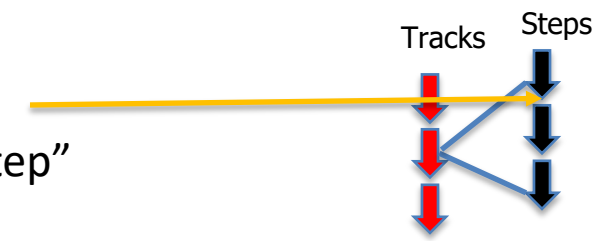
G4UserTrackingAction

- Provides user hooks to access a particle track at the beginning and end of the particle's lifetime
- virtual void BeginOfTrackingAction(const G4Track*)
  - Invoked at the beginning of a particle's lifetime (creation)
- virtual void EndOfTrackingAction(const G4Track*)
  - Invoked at the end of a particles lifetime
  - End of particle's lifetime can occur from
    - Zero kinetic energy
    - Track is explicitly killed (fStopAndKill, fKillTrackAndSecondaries)
    - Particle leaves the "world"



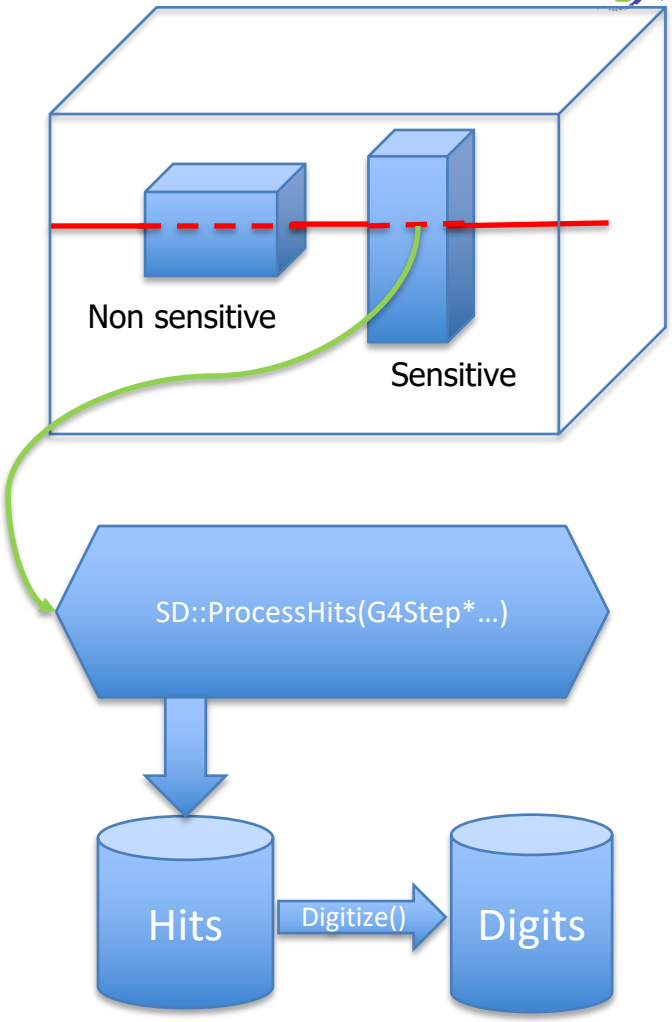
G4UserSteppingAction

- Provides user hook to a particle step
- virtual void UserSteppingAction(const G4Step*)
  - Invoked after a particle has undergone a “step”
  - A step can be defined by
    - Undergoing physical process (e.g. ionization, decay)
    - Transport step to boundary
- Typically used for custom scoring that is not supported by primitive scorers
- The most frequently called user hook
- Special attention must be paid to thread-safety when custom scoring is done here



Sensitive Detectors, Hits and Digits - Overview

- Sensitive Detector (SD) is assigned to a logical volume
- SD::ProcessHits are invoked when a step takes place in the logical volume that they are assigned to
- SDs can be used to simulate the "read-out" of your detector:
  - a way to declare a geometric element "sensitive" to the passage of particles
  - gives the user a handle to collect quantities (Hits) from these elements
    - energy deposited, position, time information
- 'Digitization' consists of converting 'Hits' into the detector response in terms of electric current & voltage signals (digits), as it would happen in the real experiment
  - same reconstruction chain can be applied for both real and simulated data



Defining a Sensitive Detector

- Sensitive detector objects are created and assigned to logical volumes in a user detector construction class in ConstructSDandField() function
- Creating SD object:

```
G4VSensitiveDetector* mySD
= new MySD("MySD", "MyHitsCollection");
```

  - Each sensitive detector object must have a unique name.
  - More than one sensitive detector instances (objects) of the same type (class) can be defined with different detector name
- Assigning to a logical volume via the volume name

```
// defined previously
// G4VSensitiveDetector* mySD = ...
SetSensitiveDetector("MyLVName", mySD);
```

Sensitive Detector Class (1/2)

- A sensitive detector is defined in a user class, MySD, derived from G4VSensitiveDetector base class
  - It defines the following user functions which are invoked by Geant4 kernel during event processing:
  - At **begin of event**: Initialize()
  - In **a step** (if in the associated volume): ProcessHits(..)
  - At **end of event**: EndOfEvent(..)

Sensitive Detector Class (2/2)

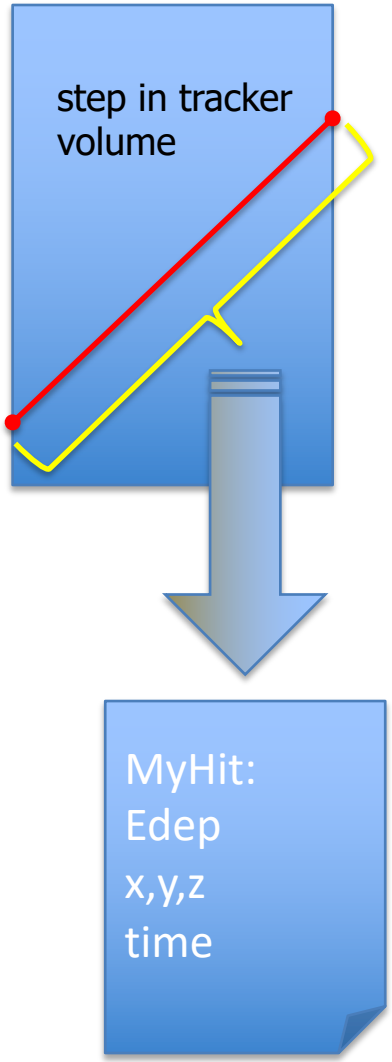
```
#include "G4VSensitiveDetector.hh"
...
class MySD : public G4VSensitiveDetector {
public:
        MySD(const G4String& name,
        const G4String& hitsCollectionName);
        virtual ~MySD();

        virtual void   Initialize(G4HCofThisEvent* hce);
        virtual G4bool ProcessHits(G4Step* step,
                                   G4TouchableHistory* history);
        virtual void   EndOfEvent(G4HCofThisEvent* hce);
};
```

User functions called by Geant4 kernel

A Hit

- Hit is a snapshot of the physical interaction of a track or an accumulation of interactions of tracks in the sensitive region of your detector

- A tracker detector typically generates a hit for every single step of every single (charged) track.
  - A tracker hit typically contains:
    - Position and time, Energy deposition of the step, Track ID
- A calorimeter detector typically generates a hit for every "cell", and accumulates energy deposition in each cell for all steps of all tracks.
  - A calorimeter hit typically contains:
    - Sum of deposited energy , Cell ID



User Hit Class

- You can store various types information by implementing your own concrete Hit class.
  - In this example we store the energy deposition of the step

MyHit.hh

```
class MyHit
{
public:
        MyHit();
        // set/get methods; eg.
        void SetEdep (G4double edep);
        G4double GetEdep() const;
private:
        // some data members; eg.
        G4double fEdep; // energy
deposit
};
```

- Typically for each information to be stored in a hit we add:

Data member	G4type fData;	G4double fEdep;
Set function	void SetData(G4type data);	void SetEdep(G4double edep):
Get function	G4type GetData() const;	G4double GetEdep() const;

Create a Hit

- A hit can be created when a step takes place in a sensitive logical volume, in a user sensitive detector function ProcessHits(..)

```
G4bool MySD::ProcessHits(G4Step* step, G4TouchableHistory* /*history*/)
{

MyHit* newHit = new MyHit();
// Get some properties from G4Step and set them to the hit
// newHit->SetXYZ();
G4double edep = step->GetTotalEnergyDeposit();
newHit->SetEdep(edep);
// ...
return true;

}
```

- Currently, returning boolean value is not used.
- The “history” will be given only if a Readout geometry is defined to this sensitive detector (the readout geometry is not presented in this course)

Hits Collections

- Many hits can be created during one event
  - Hit objects must be stored in a dedicated collection
- Geant4 provides a dedicated class, G4THitsCollection, which allows to associate the hits collections with G4Event object and can be then accessed
  - through G4Event at the end of event, to be used for analyzing an event
  - through G4SDManager during processing an event, to be used for event filtering.
- When using Geant4 hits collections, the user hit class must derive from G4VHit base class
- Users may also define their own hits collections, eg.
  - Using STL library: std::vector<MyHit>
  - Using their application framework, eg. in the context of ROOT, it can be a ROOT collection (TObjArray, TClonesArray)

User Geant4 Hit Class

- Hits collection of a concrete hit class is defined as a specialization of the G4THitsCollection template class
  - Note the analogy of G4THitsCollection<MyHit> with std::vector<MyHit>
  - To avoid long names we define a name shortcut using **typedef**

MyHit.hh

When using Geant4 hits collections, the user hit class must derive from G4VHit

```
#include "G4VHit.hh"
class MyHit : public G4VHit
{
        // the class definition as before
        // utility functions (called by Geant4)
        virtual void Draw();
        virtual void Print();
};

#include "G4THitsCollection.hh"
typedef G4THitsCollection<MyHit> MyHitsCollection;
```

Define Hits Collection (1/2)

```
void MySD::MySD(const G4String& name,
                const G4String& hitsCollectionName) :
G4VSensitiveDetector(name), fHitsCollection(0)

{
        collectionName.insert(hitsCollectionName);
}
```

- The name(s) of the hits collection(s) which is (are) handled by this sensitive detector is defined in the constructor
  - It is saved in the collectionName data member of the G4VSensitiveDetector base class
- In case your sensitive detector generates more than one kinds of hits (e.g. anode and cathode hits separately), define all collection names.

Define Hits Collection (2/2)

```
void MySD::Initialize(G4HCofThisEvent* hce)
{
  fHitsCollection = new MyHitsCollection (SensitiveDetectorName,
collectionName[0]);

  G4int hcID
   = G4SDManager::GetSDMpointer()>GetCollectionID(collectionName[0]);

  hce->AddHitsCollection(hcID, hitsCollection);
}
```

- The hits collection object is created in Initialize()
  - This method is invoked at the beginning of each event
- The collectionID, hcID, is available after this sensitive detector object is constructed and registered to G4SDManager.
  - Thus, GetCollectionID() method cannot be invoked in the constructor of this detector class.
- It can be then attached to G4HCofThisEvent object given in the argument.
  - This object is then available via G4Event object

Filling a Hits Collection (1/2)

- The hits are usually inserted in the hits collection when they are created

MySD.cc

```
void MySD::SomeFunction(...)
{
        // Create a hit
        MyHit* newHit = new MyHit();
        // Set some properties to the hit
        // newHit->SetXYZ();
        // Add the hit in the SD hits collection
        fHitsCollection->insert(newHit);
}
```

- Depending on the detector type SomeFunction() can be either Initialize() or ProcessHits()

Filling a Hits Collection (2/2)

- The way how the hits collections are filled depends on a detector type
- *A tracker detector* typically generates a hit for every single step of every single (charged) track
  - Hits are created in MySD::ProcessHits()
  - They typically contain
    - Position and time, energy deposition of the step, track ID
- *A calorimeter detector* typically generates a hit for every cell, and accumulates energy deposition in each cell for all steps of all tracks
  - Hits are created in MySD::Initialize()
  - They typically contain:
    - Sum of deposited energy, Cell ID

Digitization

- digits are created using information of hits and/or other digits by a digitizer module
- digitizer module is not associated with any volume
  - you have to implicitly invoke the Digitize() method of your concrete G4VDigitizerModule class
- G4VDigi is an abstract base class which represents a digit
  - inherit this base class and derive your own concrete digit class(es)
- G4TDigiCollection is a template class for digits collections, which is derived from the abstract base class G4VDigiCollection
- G4VDigitizerModule is an abstract base class which represents a digitizer module
  - pure virtual method Digitize() must be implemented in the concrete digitizer class
- G4DigiManager is the singleton manager class of the digitizer modules
  - concrete digitizer modules should be registered to G4DigiManager with their unique names

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->FindDigitizerModule( "/myDet/myEMdigi");
myDM->Digitize();
```

Conclusion

- User Actions and Sensitive Detectors are essential for any simulation application
  - without User Action and/or Sensitive Detectors, the simulation would run 'silently' not producing any output

- User Actions allow to
  - control the simulation flow
    - at the level of run, event, stack, track, step
  - extract information

- Sensitive Detectors (SD) are attached to specific volumes and allow to 'mimic' the readout of the real detector
  - they allow to create 'hits' which then can be 'digitized'

- Digitization modules are not associated to any volumes
  - Digitize() method needs to be invoked explicitely

Exercise

- We will be working with example B4 (examples/basic/B4) which illustrates all the items discussed in this lecture
  - go through the README file
- We will start with Variant ‘a’ where user actions are used
  - go trough the SteppingAction and EventAction to understand how the statistics is collected
  - modify the actions to collect separately the statistics for positive, negative as well as neutral particles

- We now move to variant ‘c’ where Sensitive Detectors are used
  - go through the SensitiveDetector implementation to understand how the ‘hits’ are created
  - modify the implementation to collect hits only with the energy above some threshold (for instance 1keV)