Scoring - 1

I. Hrivnacova, IPN Orsay

Credits M. Asai (SLAC), G. Folger (CERN) and others

Geant4 ED PHENIICS Tutorial,
13 - 17 May 2019, Orsay

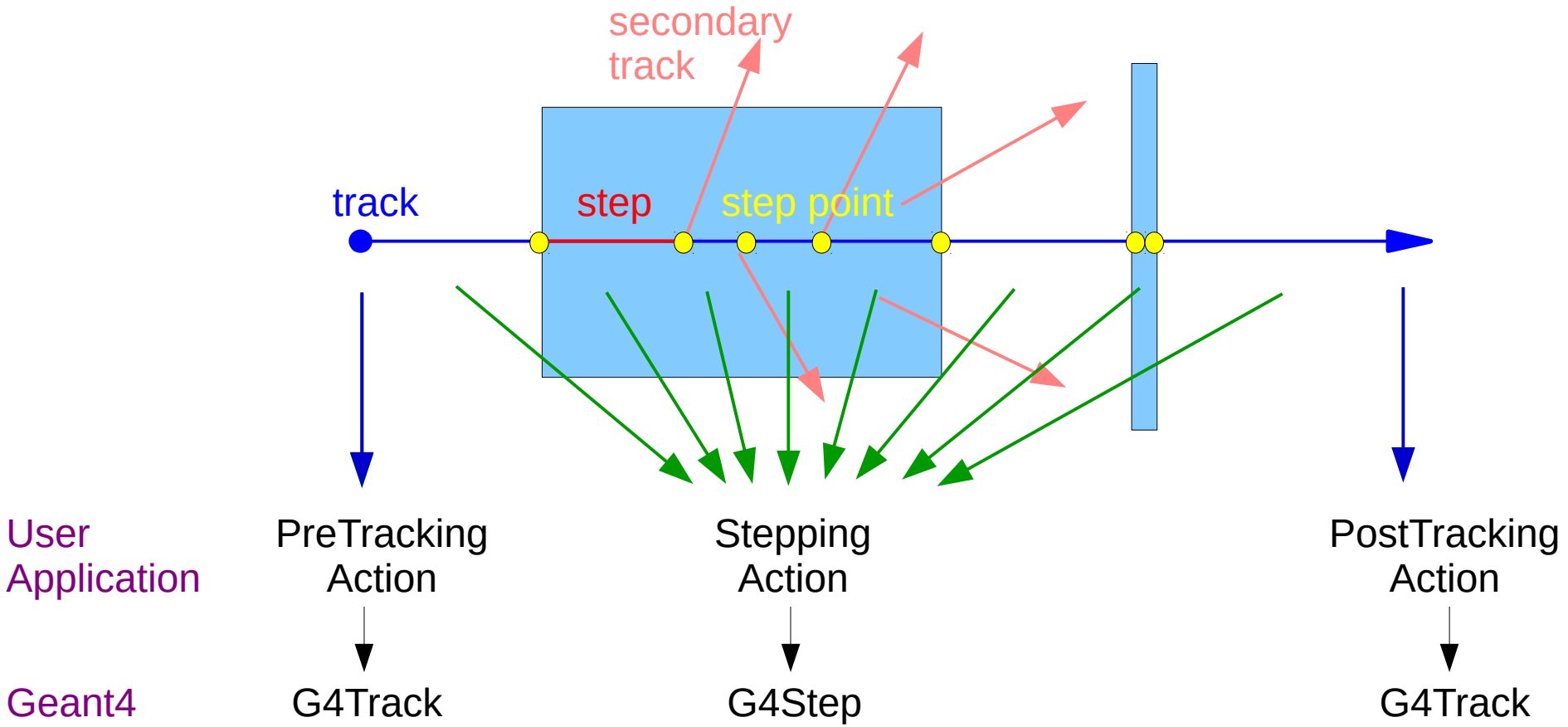
Outline

- Extracting useful information
- Sensitive detectors, hits and hits collections

Geant4 and User Application

Event Processing

User classes are called during event processing and can collect the information about tracked particles from Geant4 objects



Extracting Useful Information

- 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 and have access to “theirs” Geant4 objects:
  - G4Run, G4Event, G4Track, G4Step

Example 1

- Using G4Event information in Event action to print event number at the beginning of event

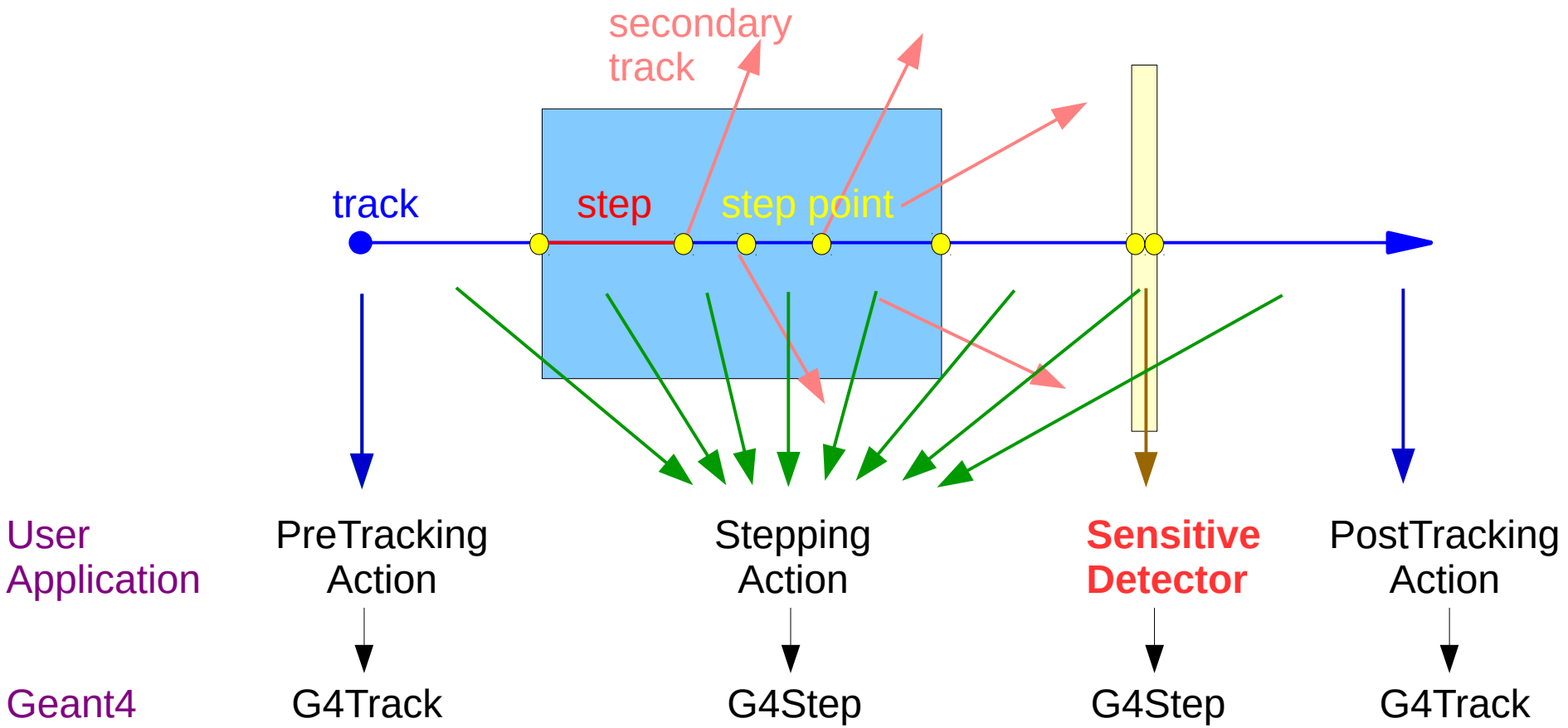
MyEventAction.cc

```
#include "MyEventAction.hh"
#include "G4Event.hh"

MyEventAction::BeginOfEventAction(const G4Event* event)
{
  // Get current event number
  G4int eventID = event->GetEventID();

  // Print this info on the screen
  G4cout << “Starting event: “ << eventID << G4endl;
}
```

Geant4 and User Application

Event Processing (2)

*A special user class, **sensitive detector**, can be attached to (a) selected volume(s) and then called during event processing*



Extracting Useful Information (2)

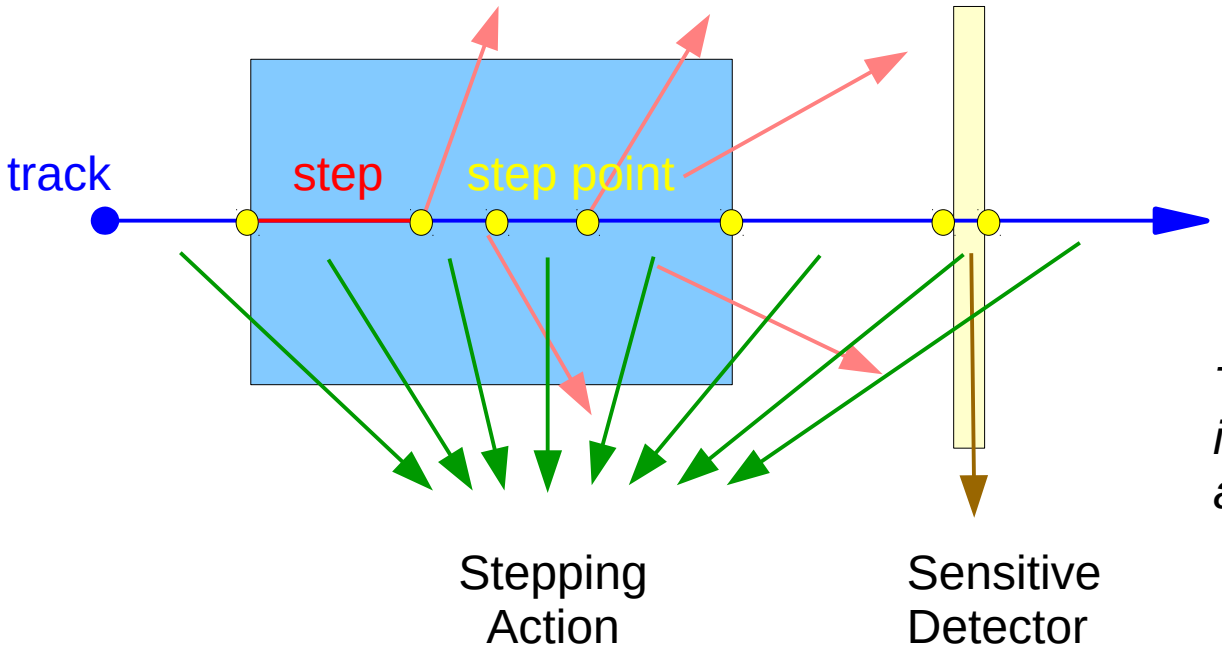
- During stepping, two user classes can be called
  - User stepping action – called in each step
  - User sensitive detector – called only when track passes a “sensitive” volume
  - Both have access to G4Step
    - Example of code where we use G4Step to access the track position

```
// Get G4Step object
G4Step* step = ...

// Get the position of the step start (pre-step point)
G4StepPoint* preStepPoint = step->GetPreStepPoint();
G4ThreeVector position = preStepPoint->GetPosition();

// Print this info on the screen
G4cout << “This step position: “ << position  << G4endl;
```

Sensitive Detectors

Sensitive Detector

- A sensitive detector is assigned to a logical volume
- The sensitive detectors are invoked when a step takes place in the logical volume that they are assigned to



The sensitive detector will be invoked all in the steps inside a sensitive volume

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:

MyDetectorConstruction.cc

```
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

- The created SD object must be registered to G4SDManager

```
// get SDManager instance
G4SDManager* sdManager = G4SDManager::GetSDMpointer();

// register this sensitive detector in SDManager
sdManager->AddNewDetector(mySD);
```

Assigning a Sensitive Detector to a Logical Volume

- Via the volume name

MyDetectorConstruction.cc

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

  - The SetSensitiveDetector function is defined in G4VUserDetectorConstruction base class (only since Geant4 10.x)

- Explicit setting to G4LogicalVolume
  - May be needed in more complex geometries

```
// defined previously
G4LogicalVolume* myLogicalVolume = ...;
G4VSensitiveDetector* mySD = ...;

// assign this sensitive detector to a logical volume
myLogicalVolume->SetSensitiveDetector(mySD);
```

Sensitive Detector Class

- 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(..)

- Note that User stepping action defines only a function invoked when processing a step

Sensitive Detector Class (2)

MySD.hh

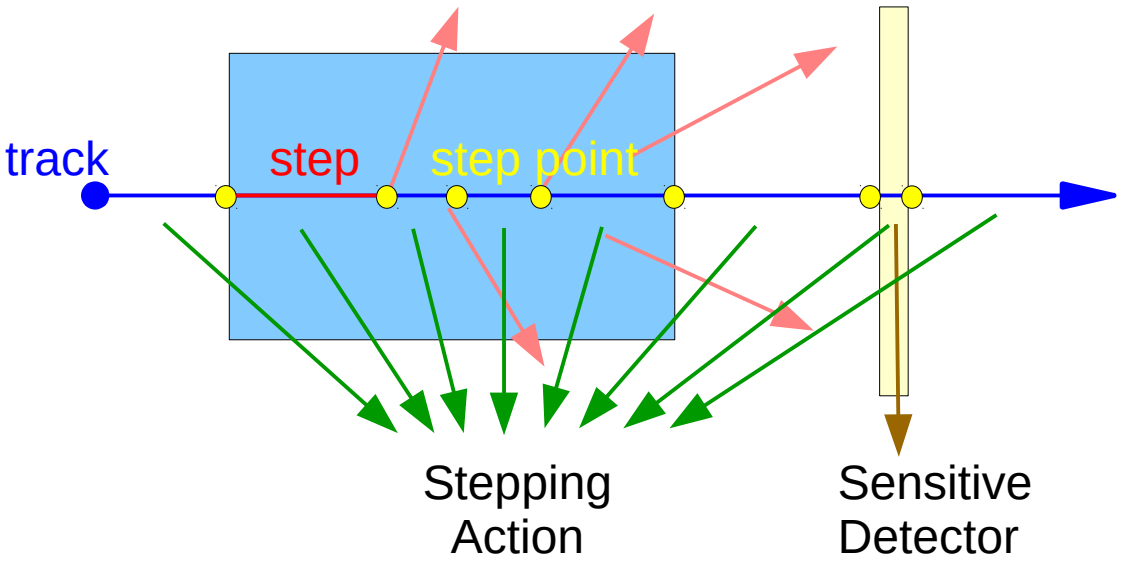
```
#include "G4VSensitiveDetector.hh"
...
class MySD : public G4VSensitiveDetector {
public:
  MySD(const G4String& name
       const G4String& hitsCollectionName););
  virtual ~MySD();

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

The user functions called by Geant4 kernel

Hits and Hits Collections

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
- Depending on your application you may be interested in various types information:
  - position and time of the step, momentum and energy of the track, energy deposition of the step, geometrical information, …



User Hit Class

MyHit.hh

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

```
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(..)

MySD.cc

```
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

```
#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;
```

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

G4Allocator

- Creation / deletion of an object is a heavy operation.
  - It may cause a performance concern, in particular for objects that are frequently instantiated / deleted like hits.
- Geant4 provides G4Allocator class which provides functions for efficient memory allocation and de-allocation
  - It allocates a chunk of memory space for objects of a certain class.
- The same pattern can be used in all user classes, its is sufficient just to put the relevant user class name

G4Allocator (2)

MyHit.hh

```
#include "G4Allocator.hh"
class MyHit : public G4VHit {
    // ...
    inline void* operator new(size_t);
    inline void  operator delete(void* hit);
    // ...
};
extern G4Allocator<MyHit>* MyHitAllocator;
inline void* MyHit::operator new(size_t) {
  return (void*)MyHitAllocator->MallocSingle();
}
inline void MyHit::operator delete(void* hit) {
  MyHitAllocator->FreeSingle((MyHit*)hit);
}
```

MyHit.cc

```
// ...
G4Allocator<MyHit>* MyHitAllocator;
// ..
```

- The pattern (in green) can be cut & pasted in your hit (and other) classes
- Then you need just to replace MyHit with your class name

Define Hits Collection (1)

MySD.cc

```
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)

MySD.cc

```
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** (`fHitsCollection`) 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

- 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)

- 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

Iterate over A Hits Collection

MySD.cc

```
void MySD::EndOfEvent(G4HCofThisEvent* /*hce*/)
{
  G4int nofHits = fHitsCollection->entries();
  G4cout << "Hits Collection: in this event there are "
         << nofHits << " hits " << G4endl;
  for ( G4int i=0; i<nofHits; ++i ) (*fHitsCollection)[i]->Print();
}
```

- The MySD::EndOfEvent() method is invoked at the end of processing an event.
  - It is invoked even if the event is aborted
  - It is invoked before UserEventAction::EndOfEventAction

Summary

- The Geant4 toolkit provides dedicated classes/tools for user scoring:
  - Sensitive detectors

  and the following which will be discussed in the next lecture
  - Geant4 scorers
  - Command-based scoring