

# Цикл обработки событий

Моделирование в Geant4 начинается с запуска (**Run**).

Свойства геометрии и физических процессов фиксируются ядром Geant4 и становятся неизменными.

Пользователь указывает, сколько событий (**Event**) он хочет моделировать в данном запуске.

Запуск начинается с момента старта первого события, а заканчивается моделированием всех вторичных частиц последнего.

События состоят из треков (**Track**), и моделирование события заканчивается с моделированием последнего трека вторичной частицы в данном событии.

Треки моделируются по одному, независимо и последовательно. Последовательность моделирования треков задается с помощью реализации стека треков.

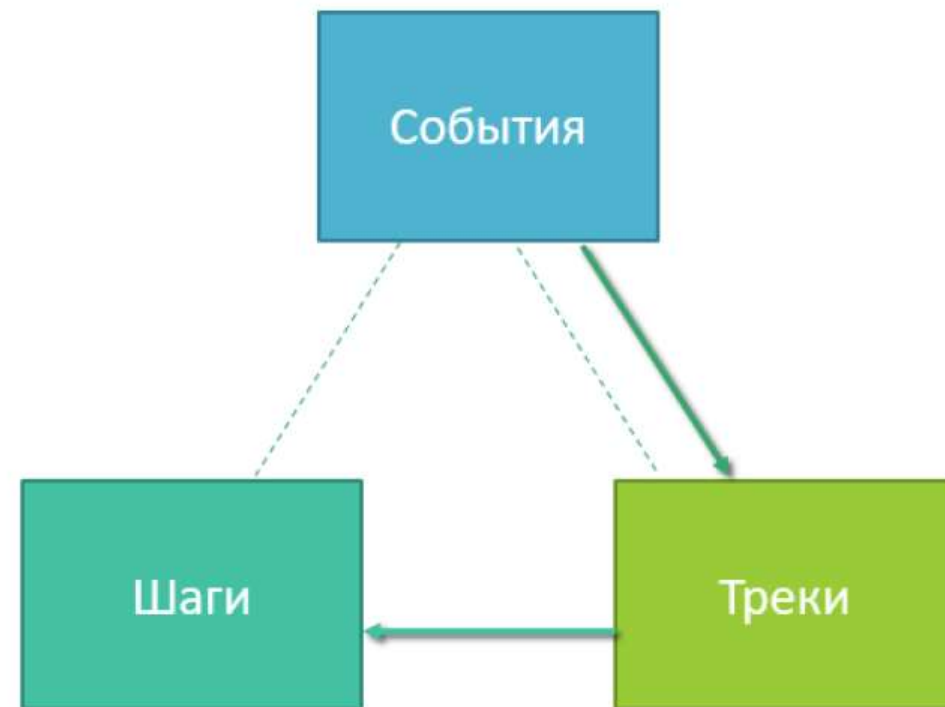
Трек содержит все свойства частицы в процессе моделирования прохождения сквозь вещество.

Каждый трек «знает» имя частицы, хранит информацию о её времени жизни и изменении кинетической энергии.

Все треки состоят из шагов (**Step**).

Шаг — это мельчайшая единица моделирования.

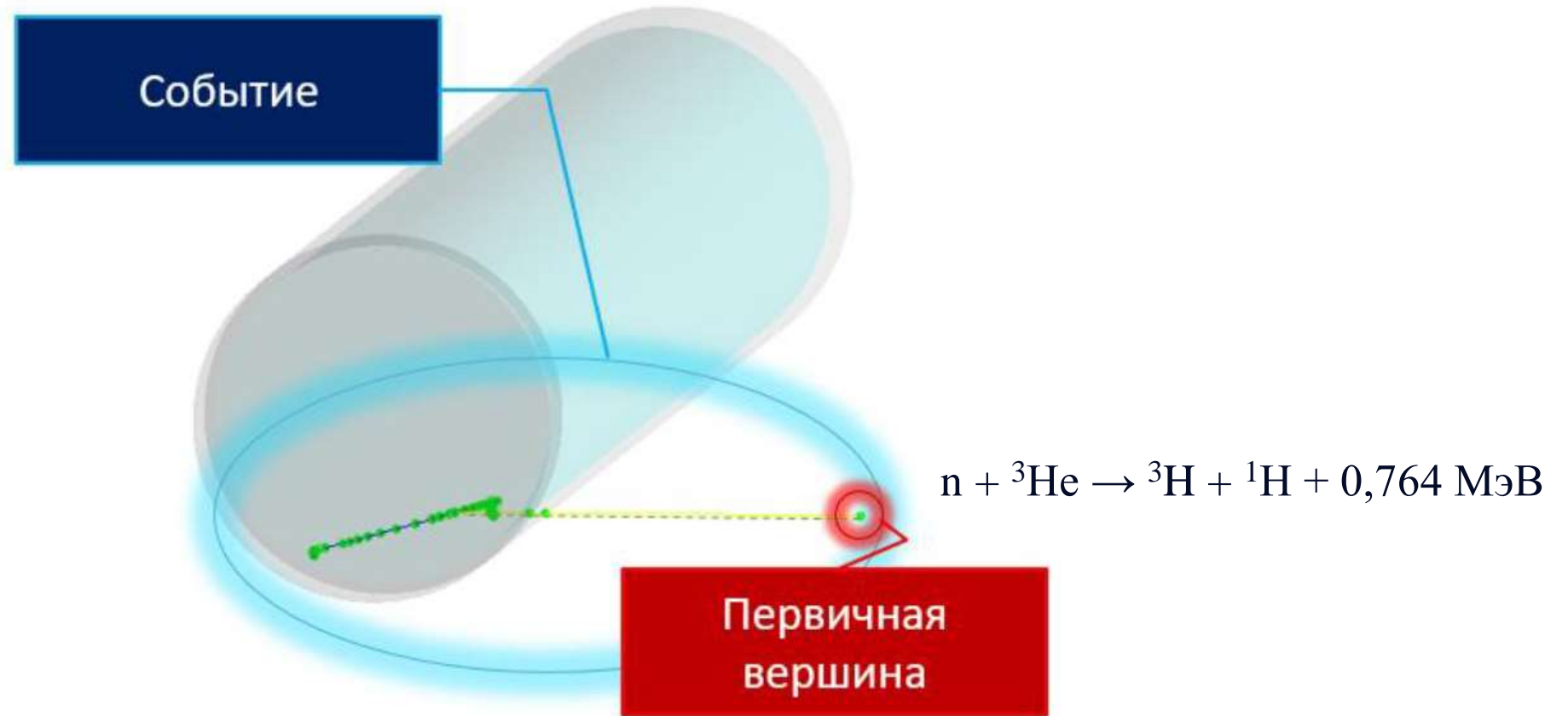
Шаг характеризует расстояние (время), за которое произошло изменение состояния частицы (переход из одного объема в другой, потери энергии на ионизацию, вылет за исследуемую область моделирования).



# Цикл обработки событий

**Событие** — представляет собой единичный цикл от зарождения первичной частицы, до окончания отслеживания последней вторичной частицы.

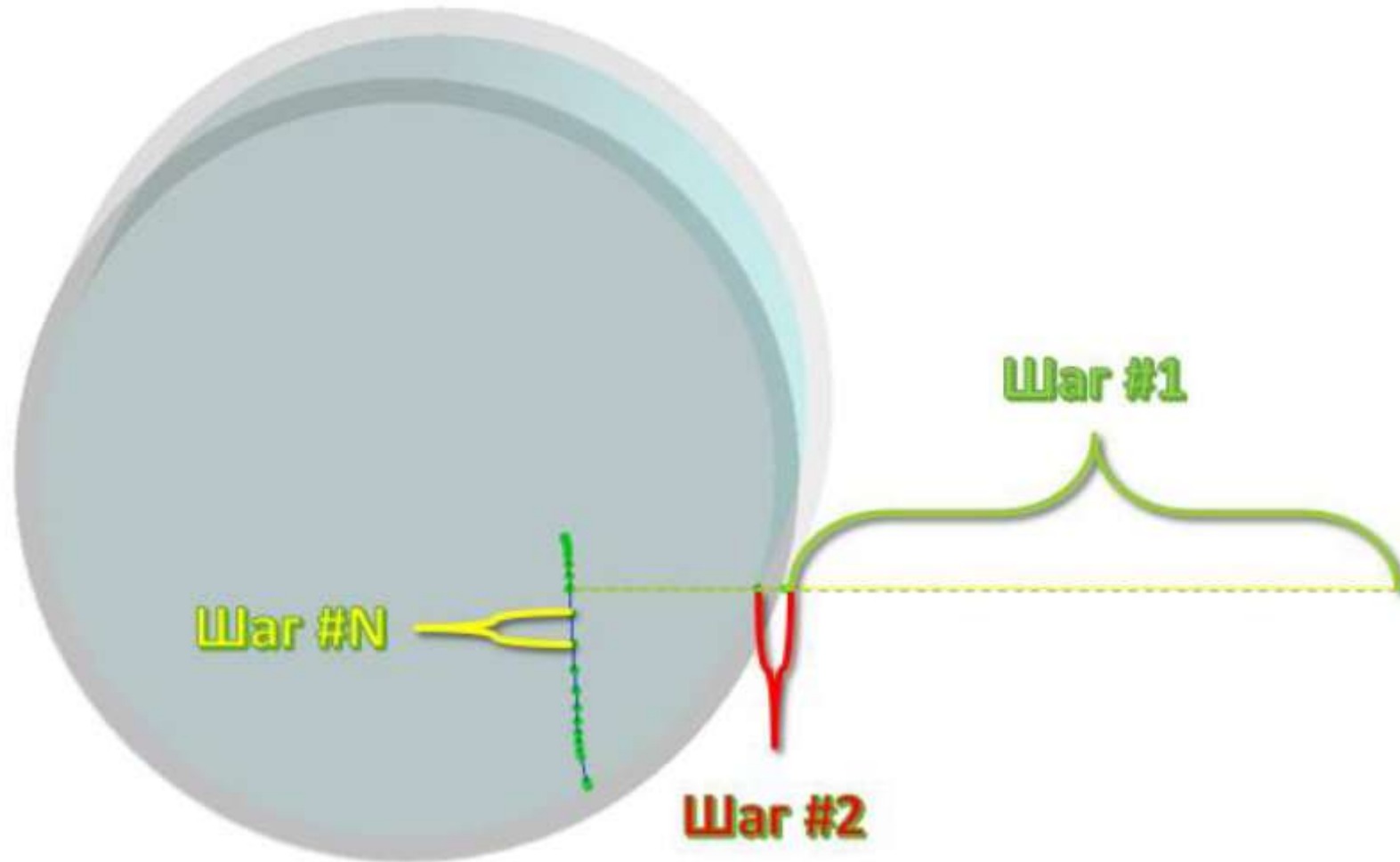
В Geant4 все события рассматриваются атомарно, т.е. независимо от остальных событий.



Нейтрон взаимодействующий в пропорциональном счетчике нейтронов с радиатором из изотопа гелия 3.  
В «событие» входят: весь трек нейтрона, треки протона и трития, а так же треки всех образованных ими вторичных частиц

# Цикл обработки событий

**Шаг** — это минимальный элемент цикла обработки событий.



# Цикл обработки событий

Пользователь напрямую не управляет ни одной из условных единиц моделирования, однако ему предоставлены классы действий, связанные со своей соответствующей единицей.

В рамках данных классов пользователь может анализировать и аккумулировать информацию в процессе моделирования, вносить некоторые корректировки в процесс моделирования.

Класс `G4VUserActionInitialization`: создание объектов классов действий.

• `G4VUserPrimaryGeneratorAction` ↔ Обязательная реализация (генерация первичной вершины)

• `G4UserRunAction`

• `G4UserEventAction`

• `G4UserStackingAction`

• `G4UserTrackingAction`

• `G4UserSteppingAction`

Чисто виртуальный метод

## Public Member Functions

	<code>G4VUserActionInitialization ()</code>
virtual	<code>~G4VUserActionInitialization ()</code>
virtual void	<code>Build () const =0</code>
virtual void	<code>BuildForMaster () const</code>
virtual <code>G4VSteppingVerbose *</code>	<code>InitializeSteppingVerbose () const</code>

## Protected Member Functions

void	<code>SetUserAction (G4VUserPrimaryGeneratorAction *) const</code>
void	<code>SetUserAction (G4UserRunAction *) const</code>
void	<code>SetUserAction (G4UserEventAction *) const</code>
void	<code>SetUserAction (G4UserStackingAction *) const</code>
void	<code>SetUserAction (G4UserTrackingAction *) const</code>
void	<code>SetUserAction (G4UserSteppingAction *) const</code>

является чисто техническим классом для определения  
порядка обработки треков

TrackStack.cc (основной файл)

DataWriter.cc (DataWriter.hh)

Loader.cc (Loader.hh)

Geometry.cc (Geometry.hh)

Action.cc (Action.hh)

PrimaryPart.cc (PrimaryPart.hh)

RunAct.cc (RunAct.hh)

EventAct.cc (EventAct.hh)

TrackAct.cc (TrackAct.hh)

StackAct.cc (StackAct.hh)

StepAct.cc (StepAct.hh)

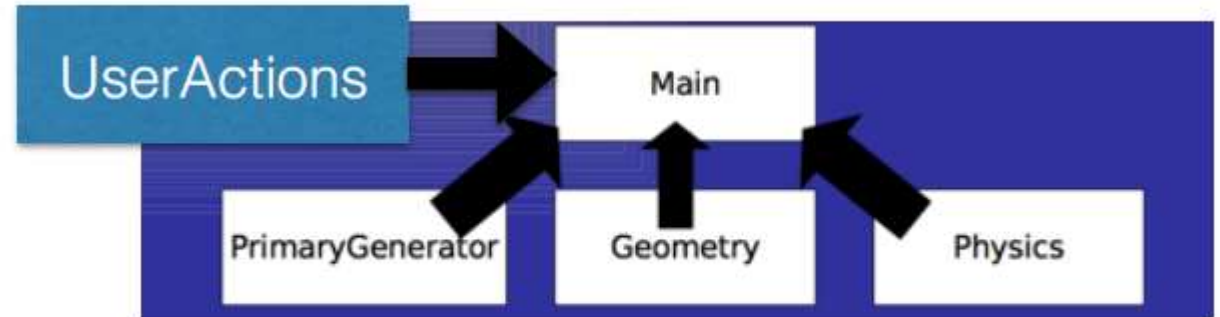
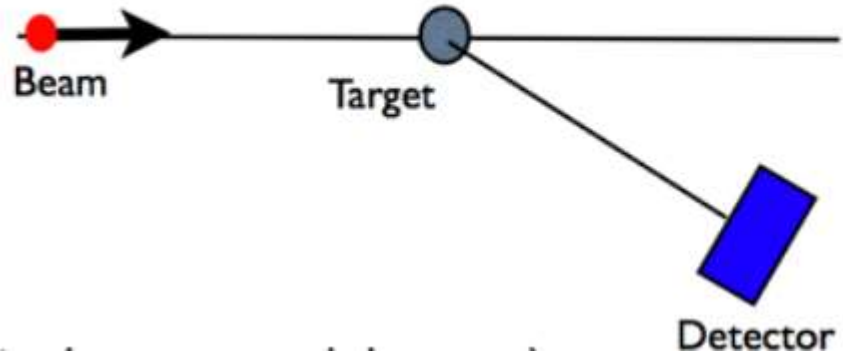
Action.hh

```
class Action: public G4VUserActionInitialization
{
    public: std::ofstream *f_act;
    public: Action(std::ofstream&);
    ~Action();
    virtual void Build() const;
};
```

Action.cc

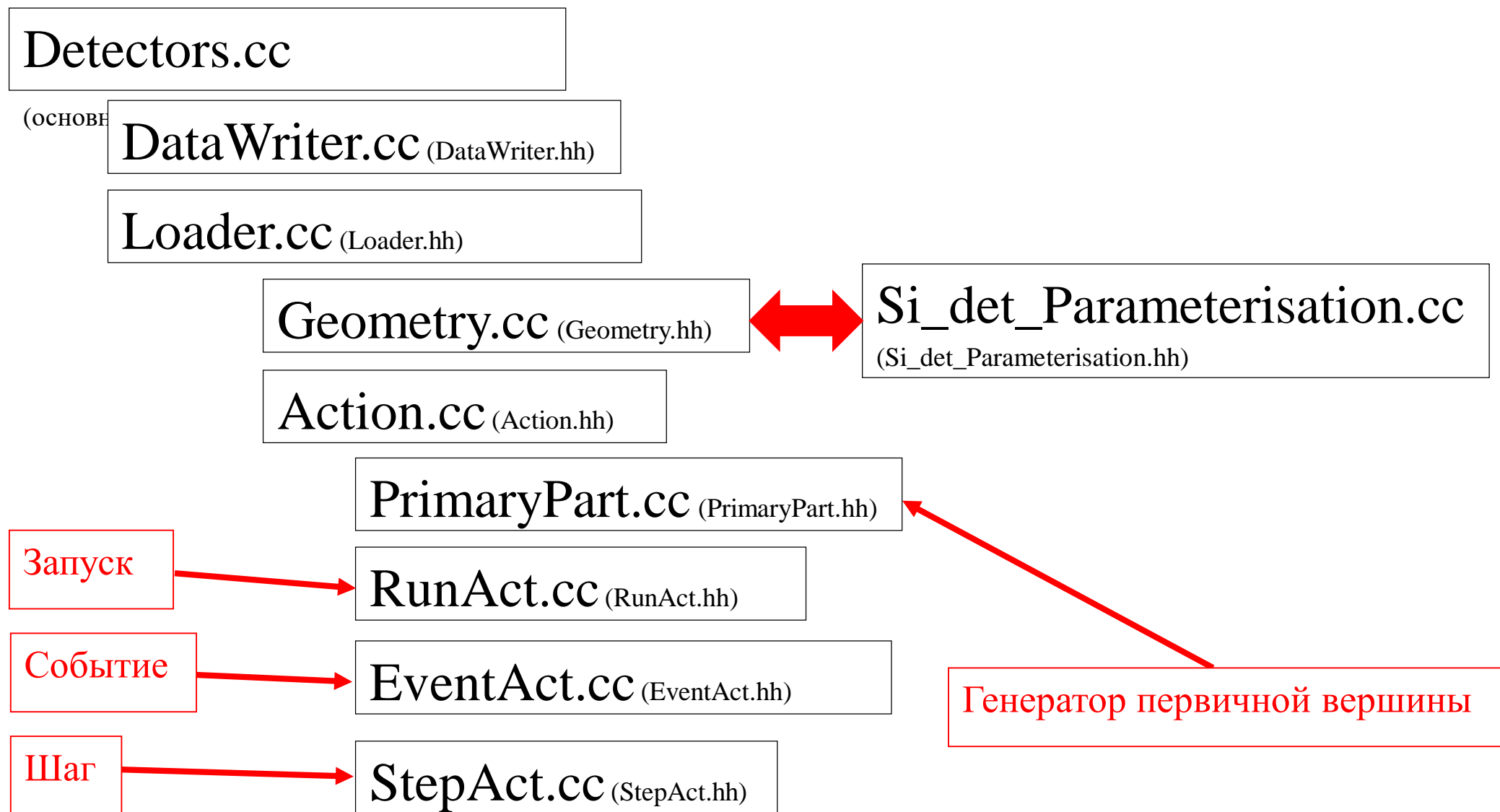
```
void Action::Build() const
{
    SetUserAction(new PrimaryPart(*this->f_act));
    SetUserAction(new RunAct(*this->f_act));
    SetUserAction(new EventAct(*this->f_act));
    SetUserAction(new TrackAct(*this->f_act));
    SetUserAction(new StackAct(*this->f_act));
    SetUserAction(new StepAct(*this->f_act));
}
```

# Цикл обработки событий



# Цикл обработки событий

проект: Detectors



# Цикл обработки событий

```
RunAct::RunAct(std::ofstream& ofsa)
```

```
{
    this->f_act=&ofsa;
    (*f_act) << std::setw(12) << "Hi from RunAct!" << std::endl;
}
```

```
RunAct::~RunAct()
```

```
{
    (*f_act) << std::setw(12) << "Bye from RunAct!" << std::endl;
}
```

```
time_t Start, End;
```

```
int RunNum = 0;
```

```
void RunAct::BeginOfRunAction(const G4Run* aRun)
```

```
{
    G4cout << "\n---Start----- Run # " << RunNum << " ----- \n" << "RunId=" << aRun->GetRunID() << G4endl;
    time(&Start);
}
```

```
void RunAct::EndOfRunAction(const G4Run* aRun)
```

```
{
    time(&End);
    G4cout << " Time spent on this Run = " << difftime(End, Start) << " seconds" << G4endl;
    G4cout << "\n---Stop----- Run # " << RunNum << " ----- \n" << "RunId=" << aRun->GetRunID() << G4endl;
    RunNum++;
}
```

Самой крупной единицей моделирования является запуск.

С началом запуска начинается цикл моделирования, а конец запуска совпадает с завершением цикла.

Класс **G4UserRunAction** является опциональным пользовательским классом «действий», связанным с запусками.



# Цикл обработки событий

**G4UserEventAction** является опциональным базовым классом, для класса пользовательских «действий» на каждом событии.

Данный класс **не осуществляет** моделирование событий (*его осуществляет экземпляр класса **G4Event***), а лишь позволяет менять параметры событий, или сохранять информацию о моделировании во время событий.

Два виртуальных метода:

```
virtual void BeginOfEventAction(const G4Event* anEvent);  
virtual void EndOfEventAction(const G4Event* anEvent);
```

Вызываются в начале (после создания первичной вершины) и конце каждого события, что можно использовать для сохранения данных.

# Цикл обработки событий

EventAct.hh

EventAct.cc

```
class EventAct : public G4UserEventAction
```

```
{
```

```
public:
```

```
    std::ofstream *f_event;
```

```
    EventAct(std::ofstream&);
```

```
    ~EventAct();
```

```
    static void Coordinates(G4ThreeVector V1, G4ThreeVector V2);
```

```
    static void AddE(G4double dE);
```

```
    static void StepLengthCounter(G4double SL);
```

```
    void BeginOfEventAction(const G4Event*);
```

```
    void EndOfEventAction(const G4Event*);
```

```
};
```

**Статические методы**

```
EventAct::EventAct(std::ofstream& ofsa)
```

```
{
```

```
    this->f_event=&ofsa;
```

```
    (*f_event) << "Hi from Event!" << std::endl;
```

```
}
```

```
EventAct::~~EventAct()
```

```
{
```

```
    (*f_event) << "Bye from Event!" << std::endl;
```

```
}
```

```
int SLcounter=0;
```

```
G4double Esum=0.;
```

```
void EventAct::Coordinates(G4ThreeVector V1, G4ThreeVector V2)
```

```
{
```

```
    G4cout << "X1=" << V1[0] * mm << " Y1=" << V1[1] * mm << " Z1=" << V1[2] * mm << G4endl;
```

```
    G4cout << "X2=" << V2[0] * mm << " Y2=" << V2[1] * mm << " Z2=" << V2[2] * mm << G4endl;
```

```
}
```

```
void EventAct::StepLengthCounter(G4double count1)
{
    SLcounter ++;
    G4cout << "Step N= " << SLcounter << "\t, Length=" << count1 * mm << G4endl;
}

void EventAct::AddE(G4double edep)
{
    Esum+=edep;
}

void EventAct::BeginOfEventAction(const G4Event * EVE)
{
    G4cout << "BeginWorks\t" << EVE->GetEventID() << G4endl;
    SLcounter = 0;
}

void EventAct::EndOfEventAction(const G4Event *EVE)
{
    (*f_event) <<"Esum=" << std::setw(12) << Esum << std::endl;
    G4cout << "EndWorks\t" << EVE->GetEventID() << G4endl;
    SLcounter=0;
    Esum=0.;
}
```

# Цикл обработки событий

**G4UserSteppingAction** является опциональным базовым классом, для класса пользовательских «действий» в конце каждого шага. Данный класс не осуществляет моделирование шагов.

stepAction.cc

```
void StepAct::UserSteppingAction(const G4Step* step)
```

```
{
```

```
  G4StepPoint* point1 = step->GetPreStepPoint();
```

```
  G4StepPoint* point2 = step->GetPostStepPoint();
```

```
  G4ThreeVector vect1, vect2;
```

```
  G4double Ekin1, Ekin2;
```

```
  vect1=point1->GetPosition();
```

```
  vect2=point2->GetPosition();
```

```
  EventAct::Coordinates(vect1,vect2);
```

```
  Ekin1=point1->GetKineticEnergy();
```

```
  Ekin2=point2->GetKineticEnergy();
```

```
  G4double StepLength = step->GetStepLength();
```

```
  EventAct::StepLengthCounter(StepLength);
```

```
  G4double edep = step->GetTotalEnergyDeposit();
```

```
  EventAct::AddE(edep);
```

```
  G4cout<<step->GetTrack()->GetDynamicParticle()->GetDefinition()->GetParticleName()<<" "<< Ekin1 * MeV
```

```
    <<" "<< Ekin2 * MeV<<" "<< StepLength * mm <<G4endl;
```

```
}
```

stepAction.hh

```
class StepAct :public G4UserSteppingAction
```

```
{
```

```
  public:
```

```
    std::ofstream *f_step;
```

```
    StepAct(std::ofstream& ofsa)
```

```
{
```

```
  this->f_step=&ofsa;
```

```
  (*f_step) << "Hi from Step!" << std::endl;
```

```
};
```

```
~StepAct()
```

```
{
```

```
  (*f_step) << "Bye from Step!" << std::endl;
```

```
};
```

```
void UserSteppingAction(const G4Step*);
```

```
};
```

Виртуальный метод  
вызывается в конце  
каждого шага.

# Цикл обработки событий

## G4StepPoint Class Reference

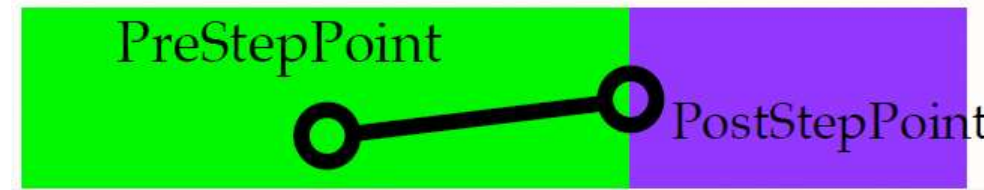
### Public Member Functions

	<b>G4StepPoint</b> ()
	<b>~G4StepPoint</b> ()
	<b>G4StepPoint</b> (const <b>G4StepPoint</b> &)
	<b>operator=</b> (const <b>G4StepPoint</b> &)
<b>G4StepPoint</b> &	<b>GetPosition</b> () const
const <b>G4ThreeVector</b> &	<b>SetPosition</b> (const <b>G4ThreeVector</b> &aValue)
void	<b>AddPosition</b> (const <b>G4ThreeVector</b> &aValue)
<b>G4double</b>	<b>GetLocalTime</b> () const
void	<b>SetLocalTime</b> (const <b>G4double</b> aValue)
void	<b>AddLocalTime</b> (const <b>G4double</b> aValue)
<b>G4double</b>	<b>GetGlobalTime</b> () const
void	<b>SetGlobalTime</b> (const <b>G4double</b> aValue)
void	<b>AddGlobalTime</b> (const <b>G4double</b> aValue)
<b>G4double</b>	<b>GetProperTime</b> () const
void	<b>SetProperTime</b> (const <b>G4double</b> aValue)
void	<b>AddProperTime</b> (const <b>G4double</b> aValue)
const <b>G4ThreeVector</b> &	<b>GetMomentumDirection</b> () const
void	<b>SetMomentumDirection</b> (const <b>G4ThreeVector</b> &aValue)
void	<b>AddMomentumDirection</b> (const <b>G4ThreeVector</b> &aValue)
<b>G4ThreeVector</b>	<b>GetMomentum</b> () const
<b>G4double</b>	<b>GetTotalEnergy</b> () const
<b>G4double</b>	<b>GetKineticEnergy</b> () const
void	<b>SetKineticEnergy</b> (const <b>G4double</b> aValue)
void	<b>AddKineticEnergy</b> (const <b>G4double</b> aValue)
<b>G4double</b>	<b>GetVelocity</b> () const
void	<b>SetVelocity</b> ( <b>G4double</b> v)
<b>G4double</b>	<b>GetBeta</b> () const
<b>G4double</b>	<b>GetGamma</b> () const
<b>G4VPhysicalVolume</b> *	<b>GetPhysicalVolume</b> () const
const <b>G4VTouchable</b> *	<b>GetTouchable</b> () const
const <b>G4TouchableHandle</b> &	<b>GetTouchableHandle</b> () const
void	<b>SetTouchableHandle</b> (const <b>G4TouchableHandle</b> &apValue)
<b>G4Material</b> *	<b>GetMaterial</b> () const
void	<b>SetMaterial</b> ( <b>G4Material</b> *)
const <b>G4MaterialCutsCouple</b> *	<b>GetMaterialCutsCouple</b> () const
void	<b>SetMaterialCutsCouple</b> (const <b>G4MaterialCutsCouple</b> *)
<b>G4VSensitiveDetector</b> *	<b>GetSensitiveDetector</b> () const

Для извлечения информации на каждом шаге используются методы класса G4Step:

**G4StepPoint\*** GetPreStepPoint() const;  
**G4StepPoint\*** GetPostStepPoint() const;

содержащие информацию о состоянии частицы в начале и конце шага, соответственно.



	void	SetSensitiveDetector (G4VSensitiveDetector *)
	G4double	GetSafety () const
	void	SetSafety (const G4double aValue)
const	G4ThreeVector &	GetPolarization () const
	void	SetPolarization (const G4ThreeVector &aValue)
	void	AddPolarization (const G4ThreeVector &aValue)
	G4StepStatus	GetStepStatus () const
	void	SetStepStatus (const G4StepStatus aValue)
const	G4VProcess *	GetProcessDefinedStep () const
	void	SetProcessDefinedStep (const G4VProcess *aValue)
	G4double	GetMass () const
	void	SetMass (G4double value)
	G4double	GetCharge () const
	void	SetCharge (G4double value)
	G4double	GetMagneticMoment () const
	void	SetMagneticMoment (G4double value)
	void	SetWeight (G4double aValue)
	G4double	GetWeight () const

# Цикл обработки событий

## Методы класса G4Step

### Public Member Functions

```
G4Step ()
~G4Step ()
G4Step (const G4Step &)
G4Step & operator= (const G4Step &)
G4Track * GetTrack () const
void SetTrack (G4Track *value)
G4StepPoint * GetPreStepPoint () const
void SetPreStepPoint (G4StepPoint *value)
G4StepPoint * GetPostStepPoint () const
void SetPostStepPoint (G4StepPoint *value)
G4double GetStepLength () const
void SetStepLength (G4double value)
G4double GetTotalEnergyDeposit () const
void SetTotalEnergyDeposit (G4double value)
G4double GetNonIonizingEnergyDeposit () const
void SetNonIonizingEnergyDeposit (G4double value)
G4SteppingControl GetControlFlag () const
void SetControlFlag (G4SteppingControl StepControlFlag)
void AddTotalEnergyDeposit (G4double value)
void ResetTotalEnergyDeposit ()
void AddNonIonizingEnergyDeposit (G4double value)
void ResetNonIonizingEnergyDeposit ()
G4bool IsFirstStepInVolume () const
G4bool IsLastStepInVolume () const
void SetFirstStepFlag ()
void ClearFirstStepFlag ()
void SetLastStepFlag ()
void ClearLastStepFlag ()
G4ThreeVector GetDeltaPosition () const
G4double GetDeltaTime () const
```

Методы, позволяющие определить изменения во времени, позиции и энергии:

```
G4double GetDeltaTime() const;
G4ThreeVector GetDeltaPosition() const;
G4double GetTotalEnergyDeposit() const;
```

доступ к треку частицы:

```
G4Track* GetTrack() const;
```

```
G4ThreeVector GetDeltaMomentum () const
G4double GetDeltaEnergy () const
void InitializeStep (G4Track *aValue)
void UpdateTrack ()
void CopyPostToPreStepPoint ()
G4Polyline * CreatePolyline () const
const std::vector< const G4Track * > * GetSecondaryInCurrentStep () const
const G4TrackVector * GetSecondary () const
G4TrackVector * GetfSecondary ()
G4TrackVector * NewSecondaryVector ()
void DeleteSecondaryVector ()
void SetSecondary (G4TrackVector *value)
void SetPointerToVectorOfAuxiliaryPoints (std::vector< G4ThreeVector > *theNewVectorPointer)
std::vector< G4ThreeVector > * GetPointerToVectorOfAuxiliaryPoints () const
```

доступ к трекам вторичных частиц:

```
const G4TrackVector* GetSecondary() const ;
```

# Цикл обработки событий

## Сбор информации с шагов в запуске

Универсальной цепочки связи между наследуемыми классами действий в базовых классах нет.

Все базовые классы действий содержат лишь конструкторы по умолчанию.

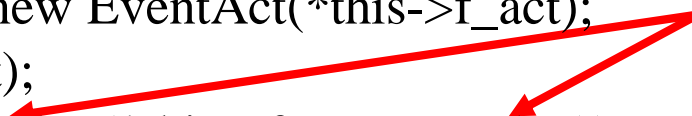
Простейшая передача информации между классами (например, от G4UserSteppingAction к G4UserEventAction) может быть реализована с использованием статических методов.

За счет особенностей наследования пользователь может сам определить, какой класс и как будет связан с другими.

проект Interact (Action.cc)

```
void Action::Build() const
{
    SetUserAction(new RunAct(*this->f_act));
    SetUserAction(new PrimaryPart(*this->f_act));
    EventAct* eventAct = new EventAct(*this->f_act);
    SetUserAction(eventAct);
    SetUserAction(new StepAct(*this->f_act,eventAct));
}
```

Изменив конструкторы классов действий таким образом, что один из них будет принимать указатель на другой, можно получить необходимую связь между ними.



# Цикл обработки событий

## проект Interact (StepAct.hh)

```
class EventAct;
```

```
class StepAct :public G4UserSteppingAction
```

```
{
```

```
public:
```

```
    std::ofstream *f_step;
```

```
    StepAct(std::ofstream& ofsa, EventAct* eventAct):event(eventAct)
```

```
    {
```

```
        this->f_step=&ofsa;
```

```
        (*f_step) << "Hi from Step!" << std::endl;
```

```
    };
```

```
    ~StepAct()
```

```
    {
```

```
        (*f_step) << "Bye from Step!" << std::endl;
```

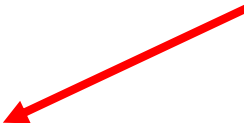
```
    };        void UserSteppingAction(const G4Step*);
```

```
private:
```


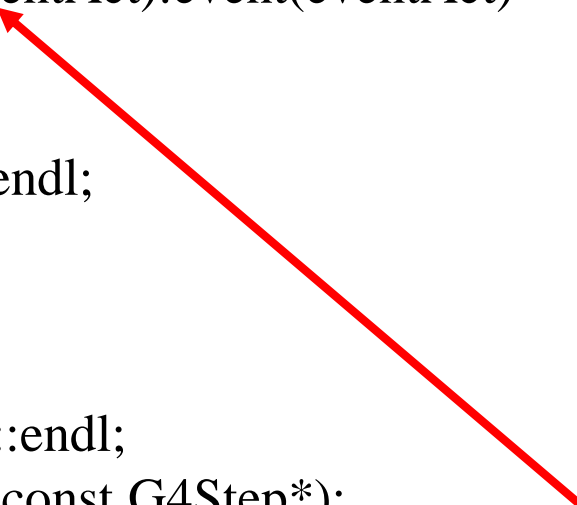
```
    EventAct* event;
```

```
};
```

инициализация члена  
event класса StepAct



обратная связь класса действий для  
шагов с экземпляром созданного  
класса действий для событий





# Цикл обработки событий

проект Interact (EventAct.cc)

проект Interact (StepAct.cc)

```
class EventAct : public G4UserEventAction
{
public:
    std::ofstream *f_event;
    EventAct(std::ofstream&);
    ~EventAct();
    void Coordinates(G4ThreeVector V1, G4ThreeVector V2);
    void AddE(G4double dE);
    void StepLengthCounter(G4double SL);
    void BeginOfEventAction(const G4Event*);
    void EndOfEventAction(const G4Event*);
};
```

**Не статические методы**

**Вызов методов созданного  
объекта event класса EventAct**

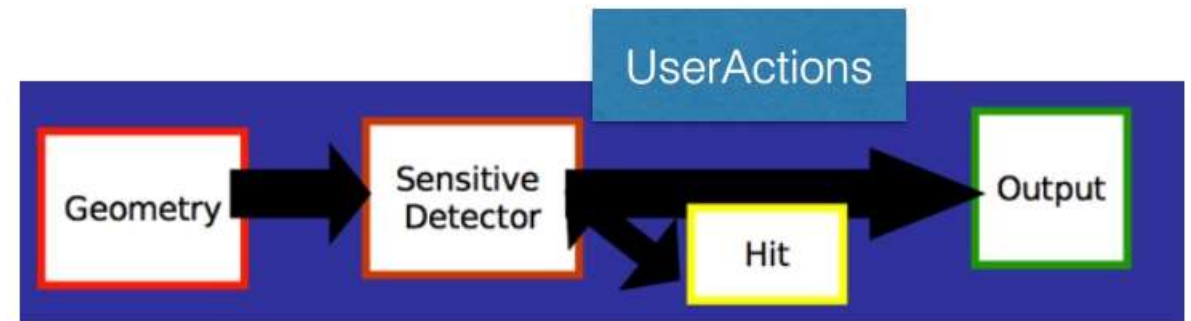
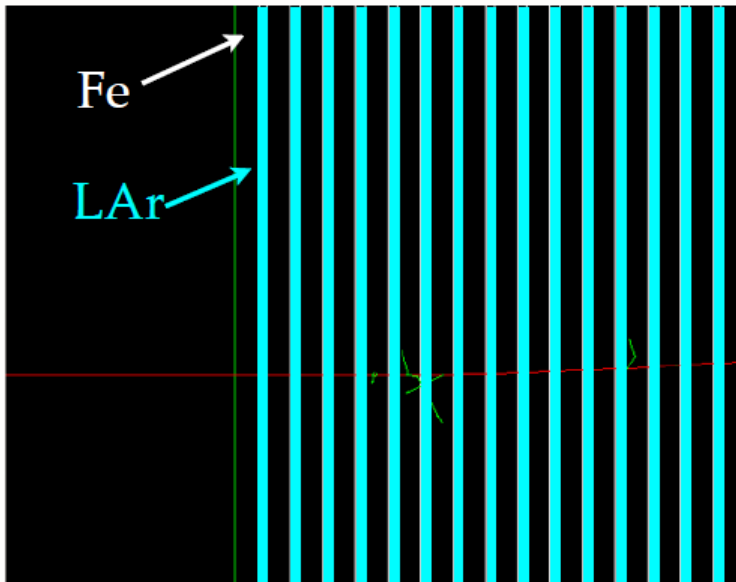
```
void StepAct::UserSteppingAction(const G4Step*
step)
{
    G4StepPoint* point1 = step->GetPreStepPoint();
    G4StepPoint* point2 = step->GetPostStepPoint();
    G4ThreeVector vect1, vect2;
    G4double Ekin1, Ekin2;
    vect1=point1->GetPosition();
    vect2=point2->GetPosition();
    event->Coordinates(vect1,vect2);
    Ekin1=point1->GetKineticEnergy();
    Ekin2=point2->GetKineticEnergy();
    G4double StepLength = step->GetStepLength();
    event->StepLengthCounter(StepLength);
    G4double edep = step->GetTotalEnergyDeposit();
    event->AddE(edep);
```

```
G4cout<<step->GetTrack()->GetDynamicParticle()->GetDefinition()->GetParticleName()<<" "<<Ekin1 * MeV<<" "<<Ekin2 *
MeV<<" "<< StepLength * mm <<G4endl;
}
```

# Создание детектирующих объёмов

- ❑ Любой логический объем в модели можно объявить детектирующим, или «чувствительным».
- ❑ При прохождении частицы через данный объем моделируется срабатывание детектора.
- ❑ Детектирующих объемов одновременно может несколько.
- ❑ Обработка срабатываний при этом может происходить по разному.
- ❑ Дополнительно можно смоделировать оцифровку сигнала и электронный отклик детектора (**ПОЗЖЕ**).

мюон, 2 ГэВ



# Создание детектирующих объёмов

## G4VSensitiveDetector Class Reference

### Public Member Functions

	<b>G4VSensitiveDetector</b> (G4String name)
	<b>G4VSensitiveDetector</b> (const <b>G4VSensitiveDetector</b> &right)
virtual	<b>~G4VSensitiveDetector</b> ()
const <b>G4VSensitiveDetector</b> &	<b>operator=</b> (const <b>G4VSensitiveDetector</b> &right)
<b>G4int</b>	<b>operator==</b> (const <b>G4VSensitiveDetector</b> &right) const
<b>G4int</b>	<b>operator!=</b> (const <b>G4VSensitiveDetector</b> &right) const
virtual void	<b>Initialize</b> (G4HCofThisEvent *)
virtual void	<b>EndOfEvent</b> (G4HCofThisEvent *)
virtual void	<b>clear</b> ()
virtual void	<b>DrawAll</b> ()
virtual void	<b>PrintAll</b> ()
<b>G4bool</b>	<b>Hit</b> (G4Step *aStep)
void	<b>SetROGeometry</b> (G4VReadOutGeometry *value)
void	<b>SetFilter</b> (G4VSDFilter *value)
<b>G4int</b>	<b>GetNumberOfCollections</b> () const
<b>G4String</b>	<b>GetCollectionName</b> (G4int id) const
void	<b>SetVerboseLevel</b> (G4int vl)
void	<b>Activate</b> (G4bool activeFlag)
<b>G4bool</b>	<b>isActive</b> () const
<b>G4String</b>	<b>GetName</b> () const
<b>G4String</b>	<b>GetPathName</b> () const
<b>G4String</b>	<b>GetFullPathName</b> () const
<b>G4VReadOutGeometry</b> *	<b>GetROGeometry</b> () const
<b>G4VSDFilter</b> *	<b>GetFilter</b> () const

### Protected Member Functions

virtual <b>G4bool</b>	<b>ProcessHits</b> (G4Step *aStep, <b>G4TouchableHistory</b> *ROhist)=0
virtual <b>G4int</b>	<b>GetCollectionID</b> (G4int i)

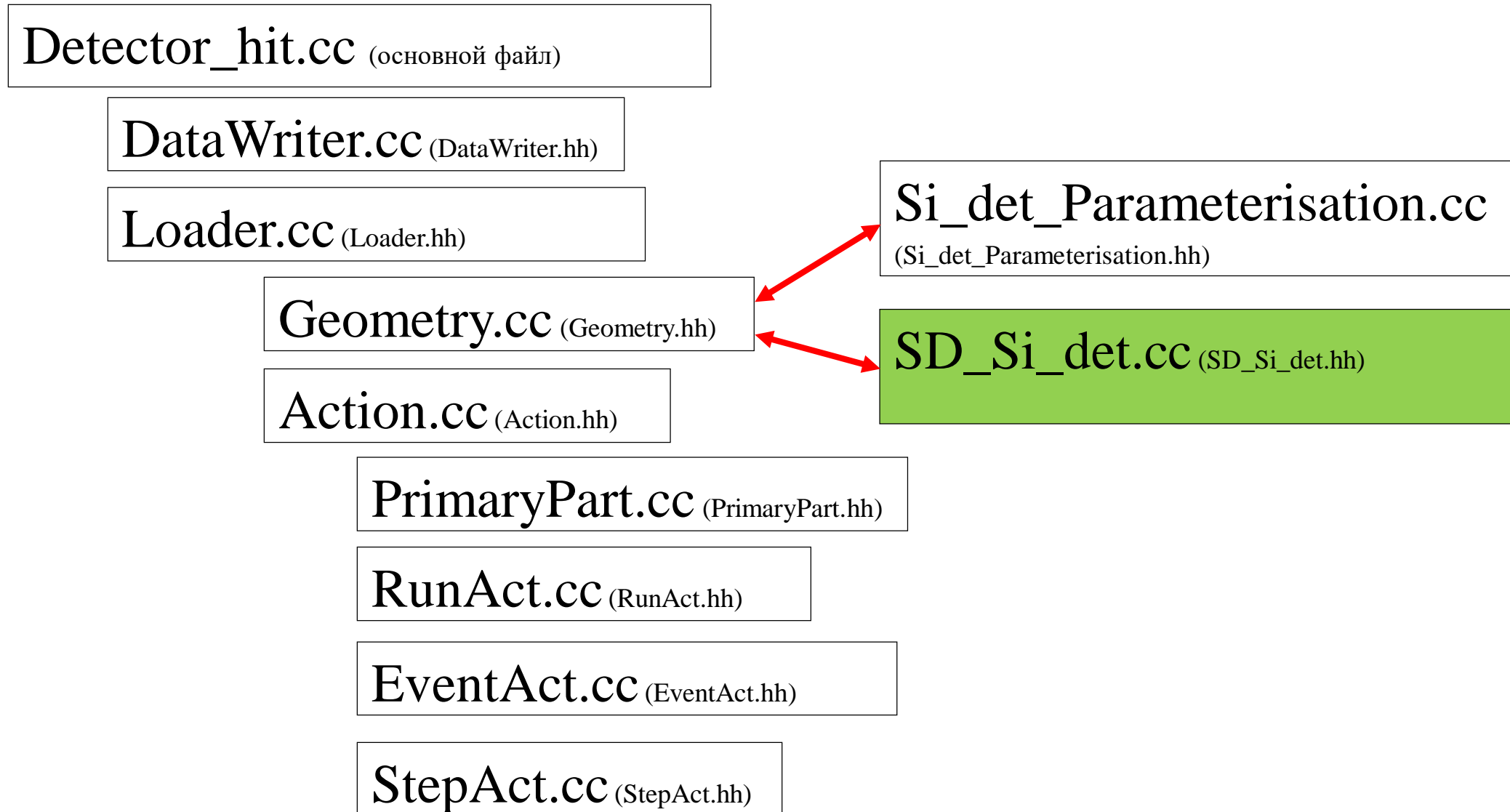
Создается класс-наследник класса G4VSensitiveDetector

Описываются обязательные методы:

- **Initialize()** вызывается в начале каждого события
- **ProcessHits()** вызывается на каждом шаге в детектирующем объеме.  
Позволяет получить информацию о характеристиках частицы в данной точке, о взаимодействии с веществом, и смоделировать срабатывание детектора.
- **EndOfEvent()** вызывается в конце события.  
Позволяет провести отбор срабатываний, и сохранить результаты.

# Создание детектирующих объёмов

проект **Detector\_hit**



# Создание детектирующих объёмов

```
G4SDManager* sdman = G4SDManager::GetSDMpointer();
SD_Si_det* sensitive_Si_det = new SD_Si_det("/mySi_det");
sdman->AddNewDetector(sensitive_Si_det);
Si_det_log->SetSensitiveDetector(sensitive_Si_det);
```

Инициализируется объект класса G4SDManager

Создание экземпляра класса для детектирующего объёма, нужно задать имя

Объект, описывающий детектирующий объем, регистрируется в менеджере

Детектирующий объем ассоциируется с логическим объемом

Sd\_Si\_det.hh

```
class SD_Si_det : public G4VSensitiveDetector
{
public:
    SD_Si_det(G4String SDname);
    ~SD_Si_det();
    G4bool ProcessHits(G4Step* astep, G4TouchableHistory*);
    void EndOfEvent(G4HCofThisEvent* HCE);
    G4double GetSumE(G4int i) const {return SumE[i];}
    void AddSumE(double e, G4int i) {SumE[i]+=e;}
private:
    std::ofstream hit_SD_Si_det[10];
    G4double SumE[10];
};
```

информация о размещении физических объёмов

коллекция откликов или срабатываний для данного события (**ПОКА** не используем)

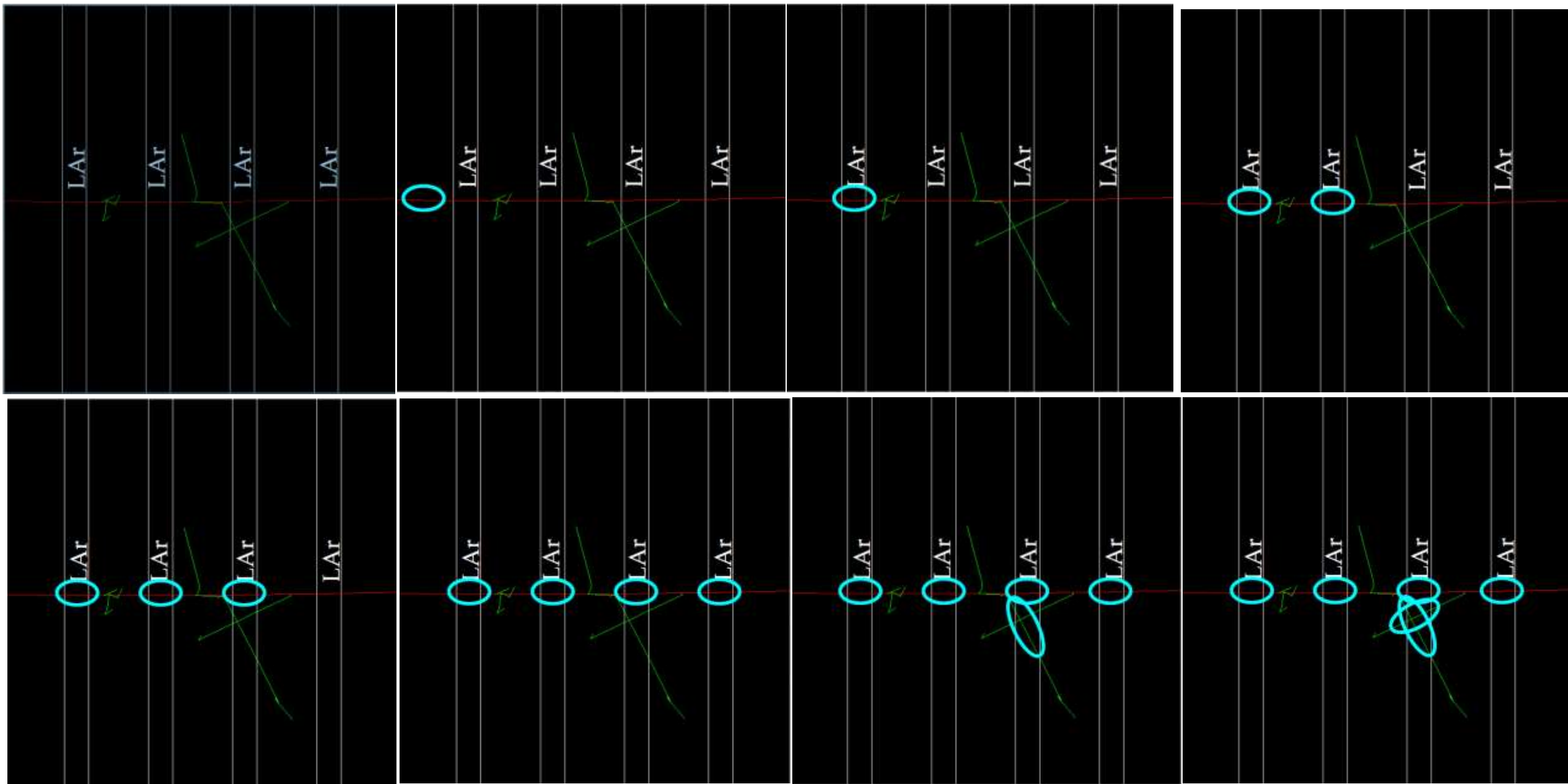
# Создание детектирующих объёмов

Sd\_Si\_det.cc

```
G4bool SD_Si_det :: ProcessHits(G4Step* step, G4TouchableHistory*)
{
  if (step->GetPostStepPoint()->GetMaterial()==G4NistManager::Instance()->FindOrBuildMaterial("G4_Si"))
  {
    G4TouchableHandle touchable = step->GetPreStepPoint()->GetTouchableHandle();
    G4int copyNo  = touchable->GetVolume(0)->GetCopyNo();
    G4double edep = 0.;
    edep          = step->GetTotalEnergyDeposit();
    this->AddSumE(edep,copyNo);
  }
  return true;
}
```

```
void SD_Si_det :: EndOfEvent(G4HCofThisEvent*)
{
  for (G4int i=0; i<10; i++)
  { hit_SD_Si_det[i] << std::setw(10) << SumE[i] << G4endl;}
  for (G4int i=0; i<10; i++) { SumE[i]=0.;}
}
```

# Создание детектирующих объёмов



# C++: Перегрузка операторов

В языке C++ определены множества операций над переменными стандартных типов, такие как +, -, \*, / и т.д.

Каждую операцию можно применить к операндам определенного типа.

Лишь ограниченное число типов непосредственно поддерживается любым языком программирования.

C и C++ не позволяют выполнять операции с комплексными числами, матрицами, строками, множествами.

**Перегрузка операторов** (operator overloading) позволяет определить для объектов классов встроенные операторы, такие как +, -, \* и т.д.

Пусть заданы множества A и B:

$A = \{ a1, a2, a3 \};$

$B = \{ a3, a4, a5 \};$

нужно выполнить операции объединения (+) и пересечения (\*) множеств.

$A + B = \{ a1, a2, a3, a4, a5 \}$

$A * B = \{ a3 \}.$

Можно определить класс Set - "множество" и определить операции над объектами этого класса, выразив их с помощью знаков операций, которые уже есть в языке C++, например, + и \*.

В результате операции + и \* можно будет использовать как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения).



# C++: Перегрузка операций

Как определить, какую функцию должен выполнить оператор: старую или новую?

По типу операндов.

А как быть с приоритетом операций?

Сохраняется определенный ранее приоритет операций.

Для распространения действия операции на новые типы данных надо определить специальную функцию, название которой содержит слово **operator** и символ перегружаемого оператора.

Функция оператора может быть определена как член класса, либо вне класса.

Перегрузить можно только те операторы, которые уже определены в C++.

Создать новые операторы нельзя.

Нельзя изменить количество операндов, их ассоциативность, приоритет.

# C++: Перегрузка операций

Если функция оператора определена как отдельная функция и не является членом класса, то количество параметров такой функции совпадает с количеством операндов оператора.

У функции, которая представляет унарный оператор (унарный минус), будет один параметр, а у функции, которая представляет бинарный оператор, - два параметра.

Если оператор принимает два операнда, то первый операнд передается первому параметру функции, а второй операнд - второму параметру.

При этом как минимум один из параметров должен представлять тип класса.

**Определение операторов в виде функций-членов класса:**

```
1 // бинарный оператор
2 ReturnType operator Op(Type right_operand);
3 // унарный оператор
4 ClassType& operator Op();
```

*ClassType* – класс, для которого определяется оператор;

*Type* – тип другого операнда;

*ReturnType* – тип возвращаемого результата, который может совпадать с типом операнда, а может и отличаться;

*Op* – операция.

# C++: Перегрузка операций

Определение операторов в виде функций, которые не являются членами класса:

```
1 // бинарный оператор
2 ReturnType operator Op(const ClassType& left_operand, Type right_operand);
3 // альтернативное определение, где класс, для которого создается оператор, представляет правый операнд
4 ReturnType operator Op(Type left_operand, const ClassType& right_operand);
5 // унарный оператор
6 ClassType& operator Op(ClassType& obj);
```

пример функции члена класса Counter, который хранит некоторое число:

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value = val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    Counter operator + (const Counter& counter) const
15    {
16        return Counter{value + counter.value};
17    }
18 private:
19     int value;
20 };
```

Результатом оператора сложения является новый объект Counter, в котором значение value равно сумме значений value обоих операндов.

оператор сложения, цель которого сложить два объекта Counter  
объект, который передается в функцию через параметр counter, будет представлять правый операнд операции.

текущий объект будет представлять левый операнд операции

```
22 int main()
23 {
24     Counter c1{20};
25     Counter c2{10};
26     Counter c3 = c1 + c2;
27     c3.print(); // Value: 30
28 }
```

После определения оператора можно складывать два объекта Counter

# C++: Перегрузка операций

пример функции вне класса:

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value = val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    int value; // к приватным переменным внешняя функция оператора не может обращаться
15 };
16 // определяем оператор сложения вне класса
17 Counter operator + (const Counter& c1, const Counter& c2)
18 {
19     return Counter{c1.value + c2.value};
20 }
21
22 int main()
23 {
24     Counter c1{20};
25     Counter c2{10};
26     Counter c3 {c1 + c2};
27     c3.print(); // Value: 30
28 }
```

внешняя функция не может обращаться к приватным полям класса, поэтому переменная value сделана публичной

бинарный оператор определяется в виде внешней функции, поэтому передаётся два параметра

первый параметр будет представлять левый операнд операции

второй параметр - правый операнд

# C++: Перегрузка операций

Функция оператора обязательно должна возвращать объект класса.

Это может быть любой объект.

Также можно определять дополнительные перегруженные функции операторов.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    Counter operator + (const Counter& counter) const
15    {
16        return Counter{value + counter.value};
17    }
18    int operator + (int number) const
19    {
20        return value + number;
21    }
22 private:
23     int value;
24 };
```

```
27 int main()
28 {
29     Counter counter{20};
30     int number = counter + 30;
31     std::cout << number << std::endl;    // 50
32 }
```

левый операнд операции должен представлять тип Counter  
правый операнд - тип int

← определена вторая версия оператора сложения, которая складывает объект Counter с числом и возвращает также число

# C++: Перегрузка операций

Какие операторы где определять?

Операторы присвоения, индексирования [], вызова (), доступа к указателю ->, инкремент ++, декремент -- определяются в виде функций-членов класса.

Операторы выделения и удаления памяти new, delete определяются в виде функций, которые не являются членами класса.

Остальные операторы можно определять в виде функций, которые не являются членами класса.

# C++: Перегрузка операций

## Операторы сравнения ==, !=, <, >

Результатом операторов сравнения, как правило, является значение типа **bool**.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    bool operator == (const Counter& counter) const
15    {
16        return value == counter.value;
17    }
18    bool operator != (const Counter& counter) const
19    {
20        return value != counter.value;
21    }
22    bool operator > (const Counter& counter) const
23    {
24        return value > counter.value;
25    }
26    bool operator < (const Counter& counter) const
27    {
28        return value < counter.value;
29    }
30 private:
31     int value;
32 };
```

Если используется простое сравнение полей класса, то для операторов == и != можно использовать специальный оператор default.

По умолчанию будут  
сравниваться все поля класса,  
для которых определен  
оператор ==.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    bool operator == (const Counter& counter) const = default;
15    bool operator != (const Counter& counter) const = default;
16 private:
17     int value;
18 };
```

```
int main()
{
    Counter c1(20);
    Counter c2(10);
    bool b1 = c1 == c2;    // false
    bool b2 = c1 > c2;    // true

    std::cout << "c1 == c2 = " << std::boolalpha << b1 << std::endl;    // c1 == c2 = false
    std::cout << "c1 > c2 = " << std::boolalpha << b2 << std::endl;    // c1 > c2 = true
}
```

# C++: Перегрузка операций

## Оператор присвоения +=

Оператор присвоения возвращает ссылку на левый операнд

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    // оператор присвоения
15    Counter& operator += (const Counter& counter)
16    {
17        value += counter.value;
18        return *this; // возвращаем ссылку на текущий объект
19    }
20 private:
21     int value;
22 };
23
24 int main()
25 {
26     Counter c1{20};
27     Counter c2{50};
28     c1 += c2;
29     c1.print();    // Value: 70
30 }
```

## Унарные операции -

Унарные операции обычно возвращают новый объект, созданный на основе имеющегося.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    // оператор унарного минуса
15    Counter operator - () const
16    {
17        return Counter{-value};
18    }
19 private:
20     int value;
21 };
22
23 int main()
24 {
25     Counter c1{20};
26     Counter c2 = -c1; // применяем оператор унарного минуса
27     c2.print();    // Value: -20
28 }
```

Операция унарного минуса возвращает новый объект Counter, значение value в котором равно значению value текущего объекта, умноженного на -1.



# C++: Перегрузка операций

## Операции инкремента ++ и декремента --

Нужно определить и префиксную, и постфиксную форму для этих операторов.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value = val;
9     }
10    void print()
11    {
12        std::cout << "Value: " << value << std::endl;
13    }
14    // префиксные операторы
15    Counter& operator++ ()
16    {
17        value += 1;
18        return *this;
19    }
20    Counter& operator-- ()
21    {
22        value -= 1;
23        return *this;
24    }
25    // постфиксные операторы
26    Counter operator++ (int)
27    {
28        Counter copy {*this};
29        ++(*this);
30        return copy;
31    }
```

Постфиксные операторы должны возвращать значение объекта до инкремента, то есть предыдущее состояние объекта. Поэтому постфиксная форма возвращает копию объекта до инкремента.

Префиксные операторы должны возвращать ссылку на текущий объект, который можно получить с помощью указателя this.

Чтобы постфиксная форма отличалась от префиксной, постфиксные версии получают дополнительный параметр типа int, который не используется.

```
32 Counter operator-- (int)
33 {
34     Counter copy {*this};
35     --(*this);
36     return copy;
37 }
38 private:
39     int value;
40 };
41
42 int main()
43 {
44     Counter c1{20};
45     Counter c2 = c1++;
46     c2.print();    // Value: 20
47     c1.print();    // Value: 21
48     --c1;
49     c1.print();    // Value: 20
50 }
```

# C++: Перегрузка операций

## Переопределение оператора <<

Оператор << принимает два аргумента: ссылку на объект потока (левый операнд) и фактическое значение для вывода (правый операнд).

Затем он возвращает новую ссылку на поток, которую можно передать при следующем вызове оператора << в цепочке.

```
3 class Counter
4 {
5 public:
6     Counter(int val)
7     {
8         value =val;
9     }
10    int getValue()const {return value;}
11 private:
12    int value;
13 };
14
15 std::ostream& operator<<(std::ostream& stream, const Counter& counter)
16 {
17     stream << "Value: ";
18     stream << counter.getValue();
19     return stream;
20 }
21
22 int main()
23 {
24     Counter counter1{20};
25     Counter counter2{50};
26     std::cout << counter1 << std::endl;    // Value: 20
27     std::cout << counter2 << std::endl;    // Value: 50
28 }
```

Стандартный выходной поток cout имеет тип std::ostream.

Поэтому первый параметр (левый операнд) представляет объект **ostream**, а второй (правый операнд) - выводимый объект Counter.

Поскольку мы не можем изменить стандартное определение std::ostream, то определяем функцию оператора, которая не является членом класса.

# C++: Перегрузка операций

## Выражение одних операторов через другие

```
3 class Counter
4 {
5 public:
6     Counter(int n)
7     {
8         value = n;
9     }
10    void print() const
11    {
12        std::cout << "value: " << value << std::endl;
13    }
14    Counter& operator+=(const Counter& counter)
15    {
16        value += counter.value;
17        return *this;
18    };
19    Counter& operator+(const Counter& counter)
20    {
21        Counter copy{ value };    // копируем данные текущего объекта
22        return copy += counter;
23    };
24 private:
25     int value;
26 };
```

оператор сложения с присвоением +=

```
28 int main()
29 {
30     Counter counter1{20};
31     Counter counter2{10};
32
33     counter1 += counter2;
34     counter1.print();    // value: 30
35     Counter counter3 {counter1 + counter2};
36     counter3.print();    // value: 40
37 }
```

В функции оператора сложения создаётся копия текущего объекта и к этой копии и аргументу применяется оператор +=