JUNO GEANT4 SCHOOL

Beijing (北京)
15-19 May 2017

(Structure of) Geant4 applications

Contents

- Geant4 design principles
- Your application: first steps

Part I:
Geant4 design principles

How to work with Geant4

- **Your model** = “normal” application written in C++
- **Geant4** = “normal” external library against which you compile and link

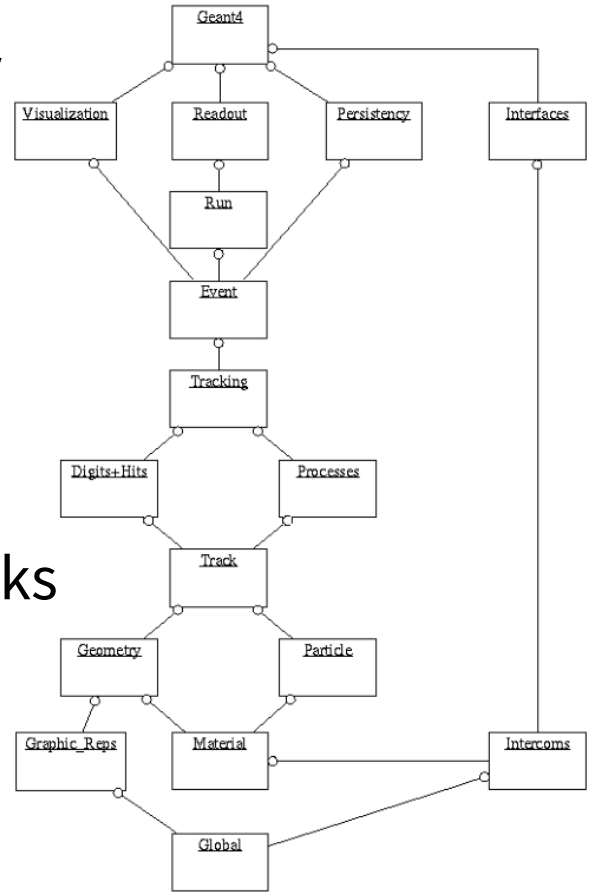
⇒ You have to:

- create an empty C++ application
- initialize Geant4 in the application `main()`
- describe the geometry, primary particles, physics and other functionality in terms of Geant4 classes
- compile the code with Geant4
- run your application

It also helps if you understand how Geant4 works!

Modular architecture

- Geant4 consists of loosely coupled modules:
  - **Run**: management of the runs
  - **Event**: management of events
  - **Tracking**: particle tracks in the geometry
  - **Processes**: physics attached to particles
  - **Particle**: elementary and other particles
  - **Geometry**: description of the detector
  - **Material**: all material properties
  - **Interfaces**: communication with user
  - **Visualization**: graphical representation of geometry & tracks
  - **...and others**



Object-oriented design

Geant4 employs many object-oriented design concepts including:

- class inheritance
- polymorphism
- method overriding
- (pure) virtual methods

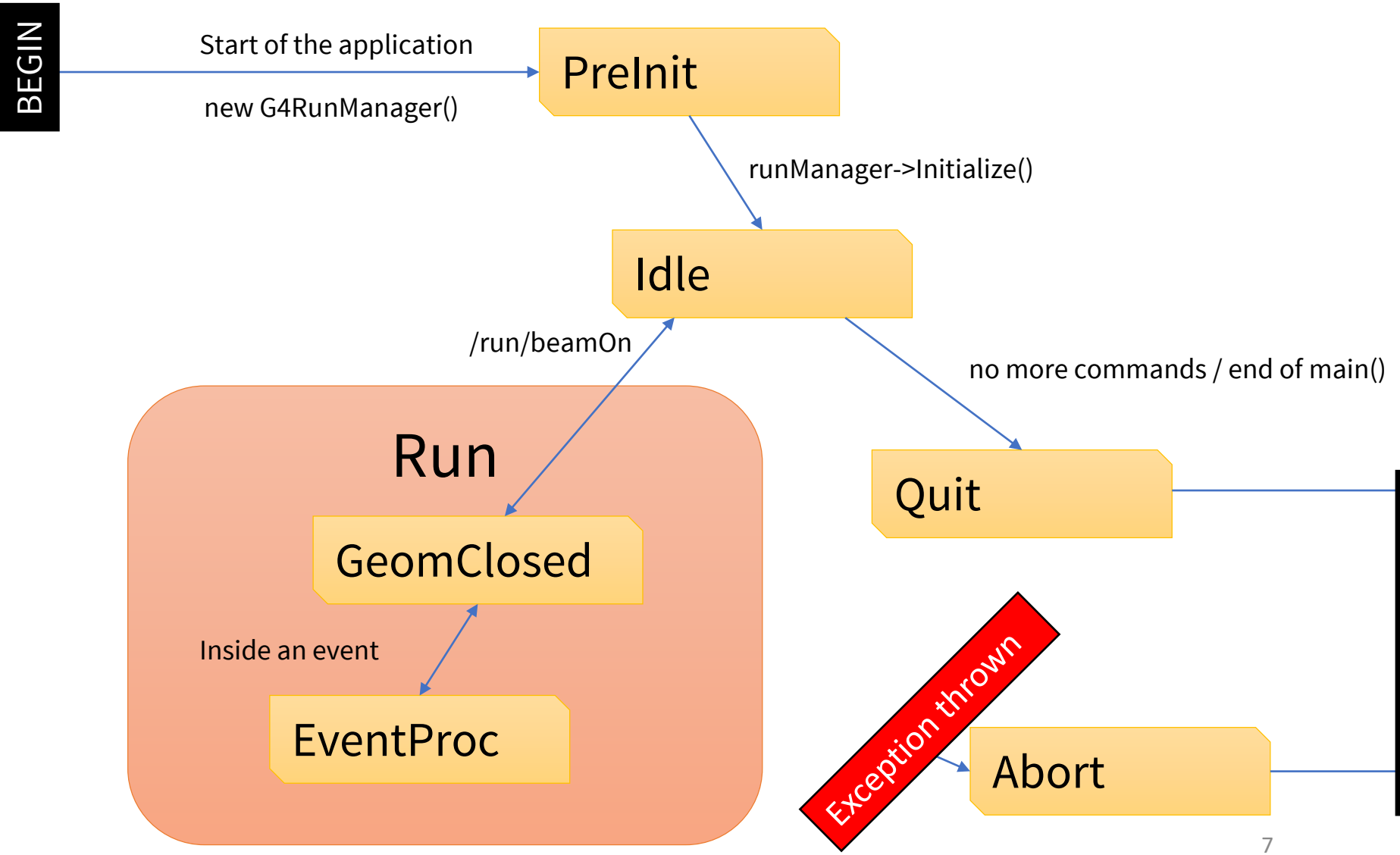
It's good to understand how C++works with these concepts.

At least a basic understanding is necessary!

To add functionality, you typically add classes inheriting from some (abstract) base class in Geant4, implementing:

- **virtual methods** that *can* be overridden
- **pure virtual methods** that *must* be overridden

Geant4 as state machine



Part II:
Your application

Note: Recommended, not enforced!

Application source structure

Official basic/B1 example:

```
2,4K  4 Dic 14:48 CMakeLists.txt
475B  4 Dic 14:48 GNUmakefile
2,8K  4 Dic 14:48 History
7,5K  4 Dic 14:48 README
4,0K  4 Dic 14:48 exampleB1.cc
226B  4 Dic 14:48 exampleB1.in
 35K  4 Dic 14:48 exampleB1.out
272B  4 Dic 14:49 include
338B  4 Dic 14:48 init_vis.mac
553B  4 Dic 14:48 run1.mac
448B  4 Dic 14:48 run2.mac
272B  4 Dic 14:49 src
3,8K  4 Dic 14:48 vis.mac
```

The text file CMakeLists.txt is the CMake script containing commands which describe how to build the exampleB1 application

contains main() for the application

Header files

```
2,2K  4 Dic 14:48 B1ActionInitialization.hh
2,4K  4 Dic 14:48 B1DetectorConstruction.hh
2,4K  4 Dic 14:48 B1EventAction.hh
2,7K  4 Dic 14:48 B1PrimaryGeneratorAction.hh
2,5K  4 Dic 14:48 B1RunAction.hh
2,4K  4 Dic 14:48 B1SteppingAction.hh
```

Source files

```
2,9K  4 Dic 14:48 B1ActionInitialization.cc
7,7K  4 Dic 14:48 B1DetectorConstruction.cc
2,6K  4 Dic 14:48 B1EventAction.cc
4,3K  4 Dic 14:48 B1PrimaryGeneratorAction.cc
5,8K  4 Dic 14:48 B1RunAction.cc
3,2K  4 Dic 14:48 B1SteppingAction.cc
```

Macro file containing the commands

How to add a new class

1) Select a class to **inherit** from (if applicable)
2) Find a good **name** for you class (no abbreviations, confusing words, otherwise inadequate)
3) Create a **header file** in include/
   - name it using the class name, with .hh extension
   - define the class (inheriting from the base)
   - declare the methods to override and other methods
4) Create a **source file** in src/
   - name it using the class name, with .hh extension
   - #include the header file
   - add definition for the class methods

Whenever you want to use it, include the header!

Typical header file

MyDetectorConstruction.hh

```
#ifndef MYDETECTORCONSTRUCTION_HH
#define MYDETECTORCONSTRUCTION_HH

#include <G4VUserDetectorConstruction.hh>

class MyDetectorConstruction
  : public G4VUserDetectorConstruction
{
public:
    MyDetectorConstruction();
    ~MyDetectorConstruction();

    G4VPhysicalVolume* Construct() override;
    void ConstructSDandField() override;

private:
    G4LogicalVolume* fLogVolume { nullptr };
};
#endif
```

Header protection (against multiple inclusions)

Include other header files (the base class + ...)

New class definition (corresponding to the file name!)

Inherit from a base class

Constructor (optional)

Virtual destructor (optional)

End the class definition with semi-colon!

A member field

Override virtual functions

G4VUserDetectorConstruction.hh

```
// ...
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField();
  // ...
}
```

Typical source file

MyDetectorConstruction.cc

```
#include "MyDetectorConstruction.hh"

#include <G4Box.hh>
#include <G4LogicalVolume.hh>
#include <G4PVPlacement.hh>

MyDetectorConstruction::MyDetectorConstruction() {
    // ...
}

~MyDetectorConstruction::MyDetectorConstruction() {
    // ...
}

G4VPhysicalVolume* MyDetectorConstruction::Construct() {
    G4Box* box = // ...
    // ...
    fLogVolume =
    return new G4PVPlacement(nullptr, {}, fLogVolume, "world", nullptr, 0, 0);
}

void MyDetectorConstruction::ConstructSDandField() {
    // ...
}
```

Including the associated header file

Include other headers

MyDetectorConstruction.hh

```
class MyDetectorConstruction
  : public G4VUserDetectorConstruction
{
public:
    MyDetectorConstruction();
    ~MyDetectorConstruction();

    G4VPhysicalVolume* Construct() override;
    void ConstructSDandField() override;

private:
    G4LogicalVolume* fLogVolume;
};
```

(and optional)

Mandatory user classes

Initialization classes

G4VUserDetectorConstruction

G4VUserPhysicsList

G4VUserActionInitialization

main()
function

Action classes

G4VUserPrimaryGeneratorAction

G4UserRunAction

G4UserEventAction

G4UserStackingAction

G4UserTrackingAction

G4UserSteppingAction

Detector construction

- Define the geometry of your model
  - All materials
  - All volumes & placements
- (Optionally) add fields
- (Optionally) define volumes for read-out (sensitive detectors)

G4VUserDetectorConstruction.hh

```
// ...
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField();
    // ...
}
```

Physics list

- Define all necessary particles
- Define all necessary processes and assign them to proper particles
- Define particles production threshold (in terms of range)

G4VUserPhysicsList.hh

```
// ...
class G4VUserPhysicsList
{
  public:
    G4VUserPhysicsList();
    virtual ~G4VUserPhysicsList();
  public:
    virtual void ConstructParticle() = 0;
    virtual void ConstructProcess() = 0;
    virtual void SetCuts();
    // ...
}
```

Primary generator action

- Define the source of simulated particles
  - particle type
  - kinematic properties
  - additional information

G4VUserPrimaryGeneratorAction.hh

```
// ...
class G4VUserPrimaryGeneratorAction
{
  public:
    G4VUserPrimaryGeneratorAction();
    virtual ~G4VUserPrimaryGeneratorAction();
  public:
    virtual void GeneratePrimaries(G4Event* anEvent) = 0;
    // ...
}
```

Other user actions

- Optional actions as hooks for different situations:
  - G4UserRunAction
  - G4UserEventAction
  - G4UserStackingAction
  - G4UserTrackingAction
  - G4UserSteppingAction
- Bind them all in G4VUserActionInitialization

G4VUserActionInitialization.hh

```
// ...
class G4VUserActionInitialization
{
  public:
    G4VUserActionInitialization ();
    virtual ~G4VUserActionInitialization ();
  public:
    virtual void Build() const = 0;
    virtual void BuildForMaster() const;
    // ...
}
```

User interaction

Communicate with your application at three levels:

- **hard-coded** application with no interaction
- **batch mode** controlled by macro files
- **interactive mode** with real-time user response
  - various terminal user interfaces
  - various graphical user interfaces

Visualization

- View and debug your geometry
- View and study the tracks
- Produce (almost) publication-ready graphics
- Export events and geometry to text files

All of that is enabled in various "**drivers**".

main() function

Geant4 **does not** provide main entry to your application, but any (C++) executable needs it!

Define it:

1) Create a source (.cc) file in the root directory of the application (name is not important)
2) Define a main function:
   - `int main()` or `int main(int argc, char** argv)`
3) Inside it:
   - initialize the run manager
   - initialize all your initialization classes
   - initialize user interface and/or visualization

Example: **very simple main()**

myApplication.cc

```
#include <G4RunManager.hh>
#include <G4UIExecutive.hh>
#include <G4VisExecutive.hh>

#include "MyDetectorConstruction.hh"
#include "MyPhysicsList.hh"
#include "MyActionInitialization.hh"

int main(int argc, char** argv) {
    G4RunManager* runManager = new G4RunManager;
    runManager->SetUserInitialization(new MyDetectorConstruction);
    runManager->SetUserInitialization(new MyPhysicsList);
    runManager->SetUserInitialization(new MyActionInitialization);
    G4VisManager* visManager = new G4VisExecutive;
    visManager->Initialize();
    G4UIExecutive* ui = new G4UIExecutive(argc, argv);
    ui->SessionStart();
    delete ui;
    delete visManager;
    delete runManager;
}
```

Include Geant4 class headers

Include your class headers

Prepare Geant4 kernel

Set up your initialization classes

Enable visualization

Enable user interface

Interact with user

Final clean-up

General recipe: application

1) Design your application... (what is supposed to do?)
2) Implement the mandatory user classes
   - detector construction
   - physics list
   - primary generator action
   - action initialization
3) Implement (optional) user action classes
   - run action, event action, stacking action, tracking action, stepping action
4) Write the main() function
   - create a run manager instance
   - register user initialization classes with the run manager
   - optionally initialize user interface and/or visualization

Note: You can actually **do** a lot **more!**