Structure of a Geant4 application

Giada Petringa
Laboratori Nazionali del Sud (LNS-INFN)

Contents

1) **Geant4 design principles**

2) **Your application: the first step**

Part I:
Geant4 design principles

How to work with Geant4

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

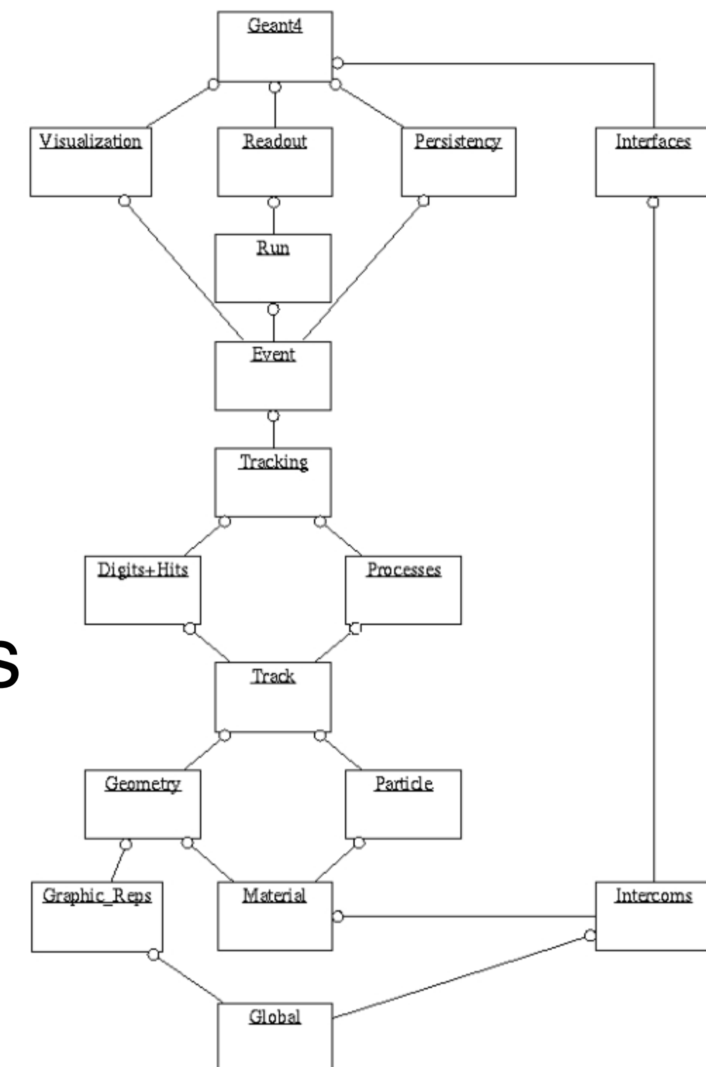
➡ **You have to:**

- 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

Modular architecture

Geant4 consists of a lot of 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**



Part II:
Your application

Application source structure

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

Macro file containing the commands

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

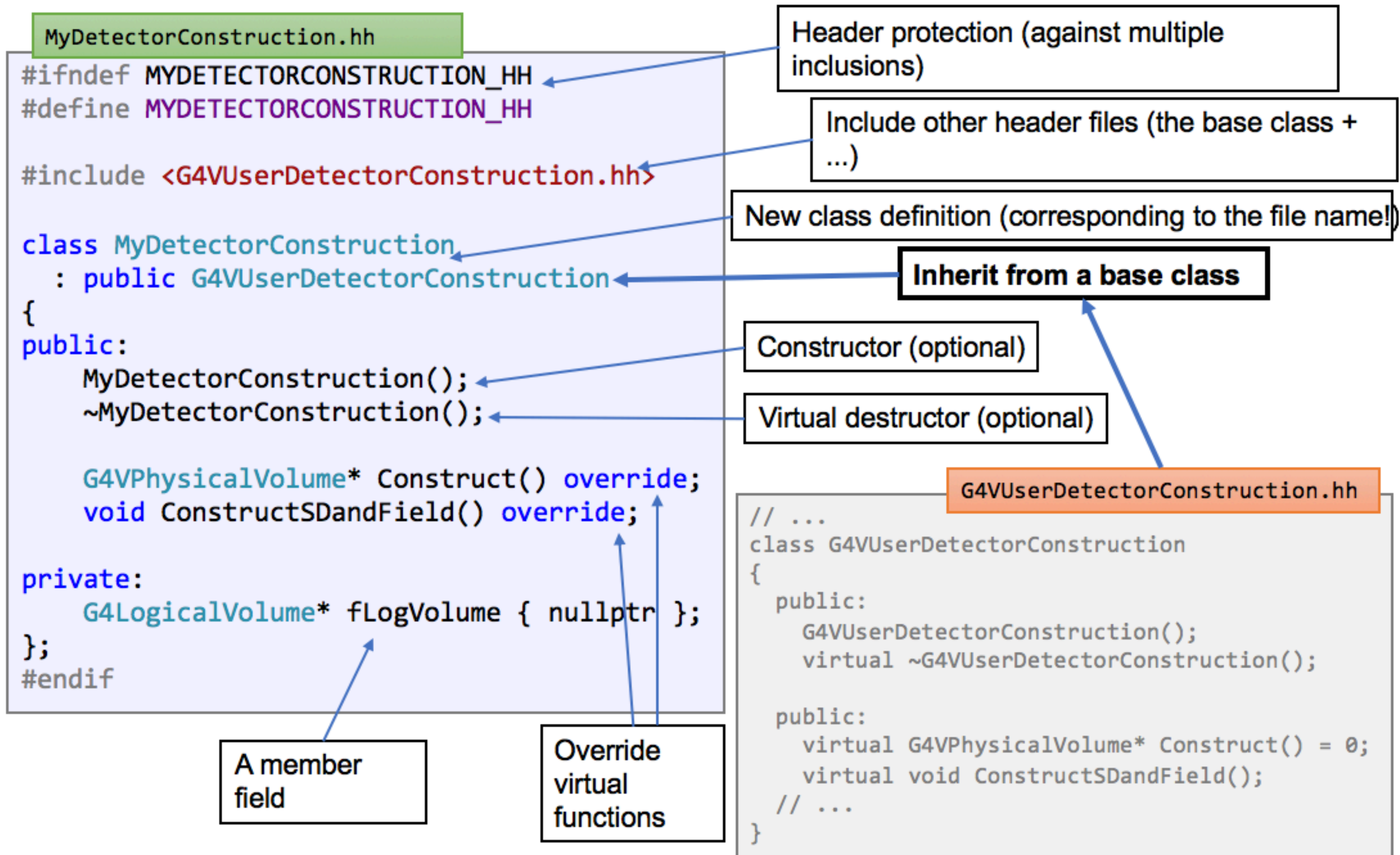
```
2,9K  4 Dic 14:48 B1ActionInitialization.cc
7,7K  4 Dic 14:48 B1DetectorConstruction.cc
2,6K  4 Dic 14:48 B1EventAction.cc
4,3K  4 Dic 14:48 B1PrimaryGeneratorAction.cc
5,8K  4 Dic 14:48 B1RunAction.cc
3,2K  4 Dic 14:48 B1SteppingAction.cc
```

How to add a new class

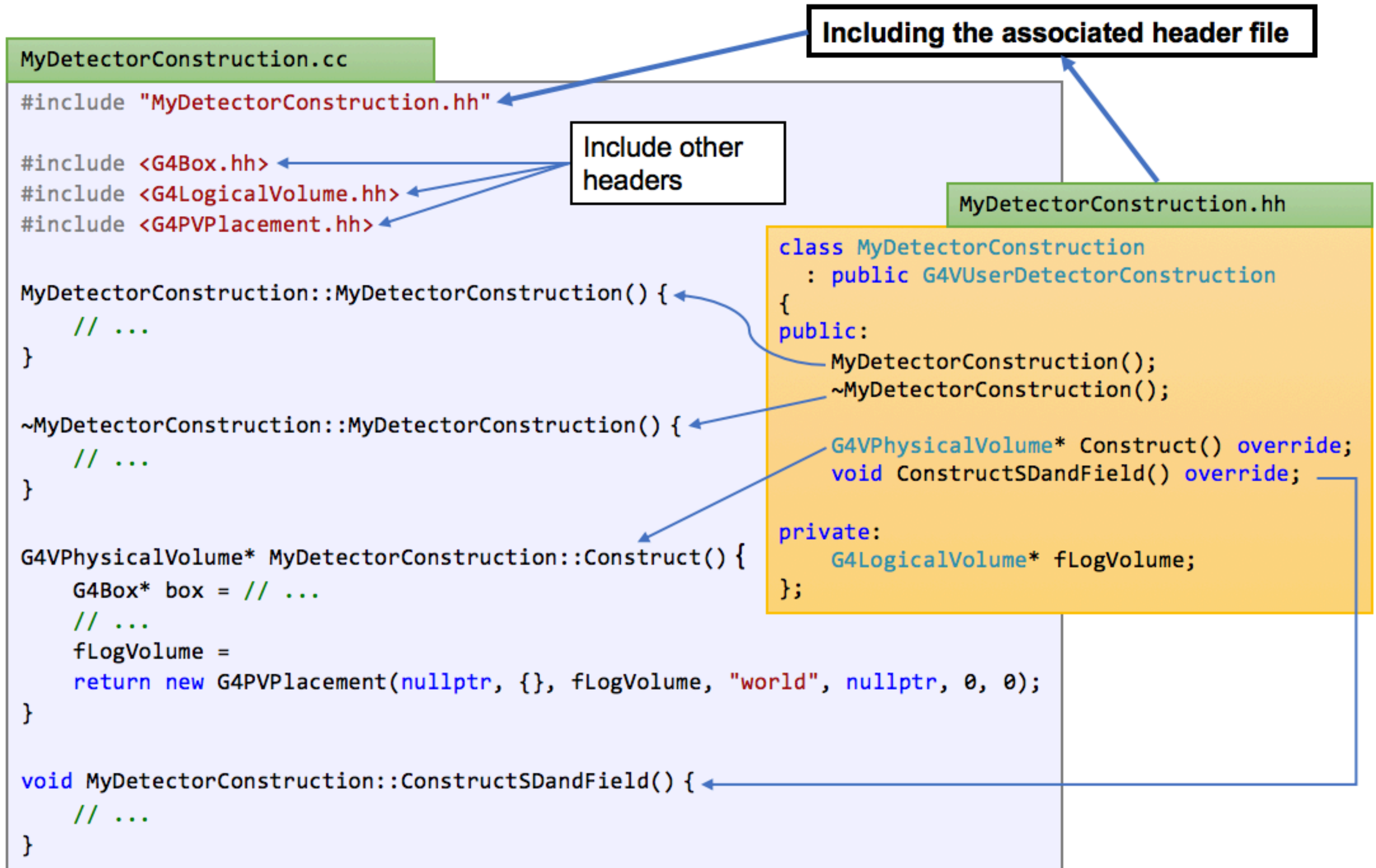
1) Select a class to **inherit** from (if applicable)
2) Find a good **name** for your class (no abbreviations, confusing words, otherwise inadequate)
3) Create a **header file** in include/
   – name it using the class name, with .hh extension
   – define the class (inheriting from the base)
   – declare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

Whenever you want to use it, include the header!

Typical header file



Typical source file



(and optional)

Mandatory user classes

Initialization classes

G4VUserDetectorConstruction

G4VUserPhysicsList

G4VUserActionInitialization

main()
function

Action classes

G4VUserPrimaryGeneratorAction

G4UserRunAction

G4UserEventAction

G4UserStackingAction

G4UserTrackingAction

G4UserSteppingAction

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;
    // ...
}
```

...more on that in
a separate talk...

Physics list

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

...more on that in 2 separate talks...

G4VUserPhysicsList.hh

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

Detector construction

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

...more on that in
3 separate talks...

G4VUserDetectorConstruction.hh

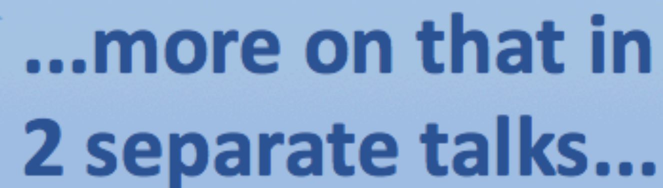
```
// ...
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

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

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



other user interaction

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

...more on that in a separate talk...

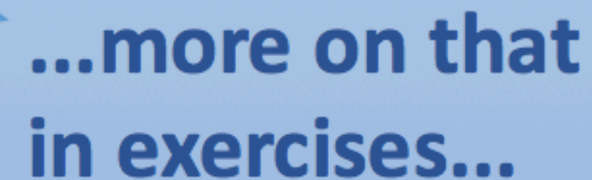
G4VUserActionInitialization.hh

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

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: **simple(st) main()**

myApplication.hh

```
#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;
}
```

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!**