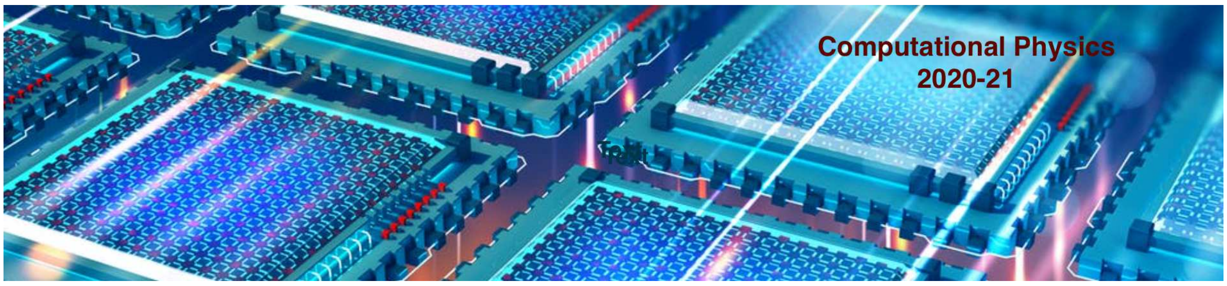
Computational Physics

numerical methods with C++ (and UNIX)

2020-21



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: fernando.barao@tecnico.ulisboa.pt

class inheritance: virtual destructors

✔ When called, the destructor of the derived class calls also the base class destructor

✔ Nevertheless, there are situations where that do not happen

- It is better to have a base class virtual destructor (is a destructor that is also a virtual function)
- The virtual destructor can ensure a proper cleanup of an object

```
class Base {
  public:
   virtual ~Base() {
   cout << "Base destructor called"
        << endl;
   }
};

class Derived: public Base {
  public:
  ~Derived() {
    cout << "Derived destructor called"
         << endl;
  }
};
```

```
// both destructors called (derived, base)
Derived *D1 = new Derived();
delete D1;

// only Base destructor is called if
// no virtual destructor on base class
Base *D2 = new Derived();
delete D2;
```

C++ classes: polymorphism

pointers to base class

- ✔ The class inheritance allows naturally a polymorphic characteristic: **a pointer to a derived class is type-compatible with a pointer to its base class**

```
Base_class *p = &Derived_class_object;
```

- ✔ A method argument can use generically a pointer to the base class for passing any derived class object

```
void func(base_class*);
```

- ✔ By default, only members of the base class are available to base class pointer

  To recover the original object a cast is needed:

```
Derived_class *p = (*Derived_class) Base_class_pointer;
```

```
#include <iostream>
using namespace std;
class Polygon {
  protected:
    float w, h;
  public:
    void set_values (float x, float y)
      { w=x; h=y; }
};
class Rectangle: public Polygon {
  public:
    float area() {return w*h;}
};
class Triangle: public Polygon {
  public:
    float area(){return w*h/2;}
};
int main() {
 Rectangle R;
 Triangle T;
  Polygon *p1 = &R;
  Polygon *p2 = &T;
  p1->set_values(4.,5.);
  p2->set_values(4.,5.);
  cout << R.area() << '\n'; // 20
  cout << T.area() << '\n'; // 10
}
```

C++ classes: polymorphism

virtual functions

- ✔ A class that declares or inherits a virtual function is called a polymorphic class
- ✔ A virtual method is a member function that can be redefined in a derived class.

  If the virtual member is =0 we are in presence of an abstract base class that cannot be instantiated by itself!

```
// virtual function
virtual string GetBranch();

//pure virtual function
virtual string GetBranch() = 0;
```

- ✔ **virtual** declarations, allow base class pointers to derived classes, catch the right method (the one redefined on derived class)

  polymorphism working!!!!!

```
#include <iostream>
using namespace std;
class Polygon {
  protected:
    float w, h;
  public:
    void set_values (float x, float y)
    { w=x; h=y; }
    virtual float area() {return 0;}
};
class Rectangle: public Polygon {
  public:
    float area() {return w*h;}
};
class Triangle: public Polygon {
  public:
    float area(){return w*h/2;}
};
int main () {
  Rectangle R; Triangle T; Polygon P;
  Polygon *p1 = &R; Polygon *p2 = &T;
  Polygon *p3 = &P;
  p1->set_values (4,5); p2->set_values (4,5);
  p3->set_values (4,5);
  cout << p1->area() << '\n'; // 20
  cout << p2->area() << '\n'; // 10
  cout << p3->area() << '\n'; // 0
}
```

C++ inheritance: example

Playing with inheritance and polymorphism: C -> B -> A

class A

```cpp
#include <iostream>
using namespace std;
#include <cstdio>
#include <cmath>

class A {
  public:
   A(float fx=0.0) : x(fx) {
     cout << __PRETTY_FUNCTION__ << "Constructor A called" << endl;}
   virtual ~A() {
     cout << __PRETTY_FUNCTION__ << "destructor A called" << endl;}
   virtual void Print() {
     cout << __PRETTY_FUNCTION__ << "print A (x=" << x << ")" << endl;}
   virtual float Norma() { return fabs(x); }
   void NormaPrint() { cout << "Norma=" << this->Norma() << endl;}

   protected:
    float x;
};
```

C++ inheritance example

class B

```cpp
class B: public A {
  public:
     // B(float fx=0.0) : x(fx) {;} // NOT CORRECT
     // B(float fx=0.0) { x=fx; }   // CORRECT but not fast
     B(float fx=0.0, float y_=0) : A(fx), y(y_) {;}     // CORRECT and FAST
        cout << __PRETTY_FUNCTION__ << "Constructor B called" << endl;}
     ~B() {
        cout << __PRETTY_FUNCTION__ << "destructor B called" << endl;}
      void Print() {
        cout << __PRETTY_FUNCTION__ << " print B (x,y=" << x << y<< ")" << endl;}
      float Norma() { return fabs(x)+fabs(y); }
    protected:
      float y;
};
```

class B **inherits from** ***class A***

Print() **method redefined**

Norma() **method redefined**

C++ inheritance example

class C

```cpp
class C: public B {
  public:
    C(float fx=0.0, float fy=0) : B(fx,fy) {
       cout << "Constructor C called" << endl;}
    ~C() {
       cout << "destructor C called" << endl;}
    void Print() {
       cout << "print C (x=" << x << ")" << endl;}
   float Norma() {
       return x*y;}
};
```

class Tools

a *fPrint()* method is defined with a base class as argument! Which Print() method will be called?

```cpp
class Tools {
  public:
    static void fPrint(string s) {
      cout << s << endl;}
    static void fPrint(A* p) {
      p->Print();}
 };
```

C++ inheritance example

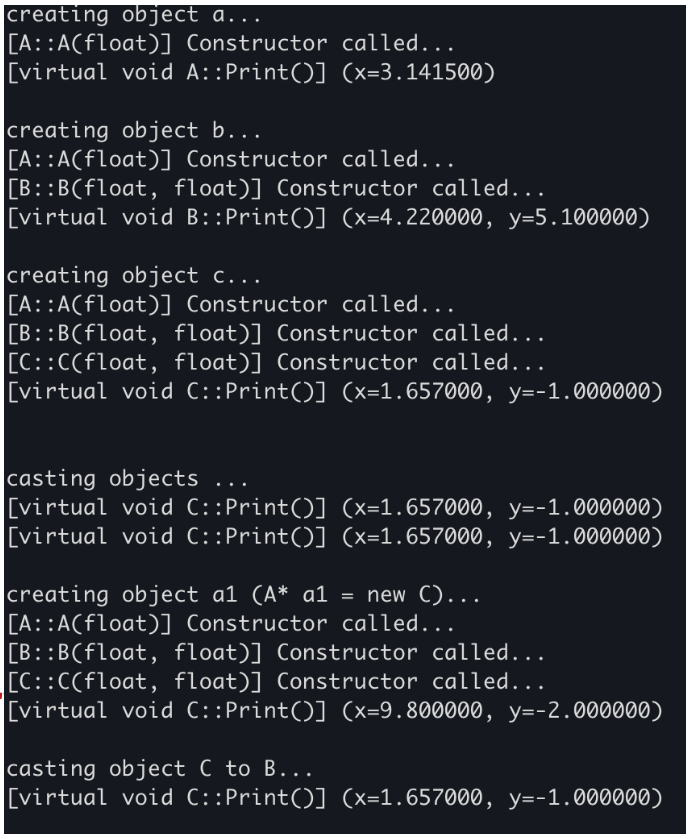
```cpp
#include "A.h"
#include "B.h"
#include "C.h"
#include "Tools.h"
#include <cstdlib>

int main() {
  printf("creating object a...\n");
  A* a = new A(3.1415); a->Print();
  printf("\ncreating object b...\n");
  B* b = new B(4.22, 5.1); b->Print();
  printf("\ncreating object c...\n");
  C* c = new C(1.657, -1.0); c->Print();

  printf("\ncasting objects ...\n");
  A* ca1 = c; ca1->Print();
  A* ca2 = (A*)c; ca2->Print();

  printf("creating object a1 (A* a1 = new C)...\n
  A* a1 = new C(9.8, -2.0); a1->Print();

  printf("casting object C to B...\n");
  B* b1 = (B*)c; b1->Print();
```



```
creating object a...
[A::A(float)] Constructor called...
[virtual void A::Print()] (x=3.141500)

creating object b...
[A::A(float)] Constructor called...
[B::B(float, float)] Constructor called...
[virtual void B::Print()] (x=4.220000, y=5.100000)

creating object c...
[A::A(float)] Constructor called...
[B::B(float, float)] Constructor called...
[C::C(float, float)] Constructor called...
[virtual void C::Print()] (x=1.657000, y=-1.000000)


casting objects ...
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[virtual void C::Print()] (x=1.657000, y=-1.000000)

creating object a1 (A* a1 = new C)...
[A::A(float)] Constructor called...
[B::B(float, float)] Constructor called...
[C::C(float, float)] Constructor called...
[virtual void C::Print()] (x=9.800000, y=-2.000000)

casting object C to B...
[virtual void C::Print()] (x=1.657000, y=-1.000000)
```

C++ inheritance example

```cpp
  printf("\ncreating 100 A pointers and"
         " casting objects...\n");
  int N=0;
  A** buf = new A*[100];
  buf[N] = a;  N++;
  buf[N] = b;  N++;
  buf[N] = c;  N++;
  buf[N] = a1; N++;

  A* d; Tools::fPrint(d=new C);
  buf[N] = d; N++;
  printf("buffer: nb of pointers stored =%d\n",N);

  for (int i=0; i<N; i++) {
    buf[i]->Print();
  }

  for (int i=0; i<N; i++) {
    Tools::fPrint(buf[i]);
  }

  for (int i=0; i<N; i++) {
    buf[i]->NormaPrint();
  }
```

```
creating 100 A pointers and casting objects...
[A::A(float)] Constructor called...
[B::B(float, float)] Constructor called...
[C::C(float, float)] Constructor called...
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=0.000000, y=0.000000)
buffer: number of pointers stored =5


print all objects (5) --------
[virtual void A::Print()] (x=3.141500)
[virtual void B::Print()] (x=4.220000, y=5.100000)
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[virtual void C::Print()] (x=9.800000, y=-2.000000)
[virtual void C::Print()] (x=0.000000, y=0.000000)


print all objects (5) with Tools::fPrint(A*) --------
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void A::Print()] (x=3.141500)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void B::Print()] (x=4.220000, y=5.100000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=9.800000, y=-2.000000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=0.000000, y=0.000000)


calling NormaPrint from class A that uses this --------
[virtual float A::Norma()] Norma called (x=3.141500)... 3.141500
[virtual float B::Norma()] Norma called (x=4.220000, y=5.100000)... 9.320000
[virtual float C::Norma()] Norma called (x=1.657000, y=-1.000000)... -1.657000
[virtual float C::Norma()] Norma called (x=9.800000, y=-2.000000)... -19.600000
[virtual float C::Norma()] Norma called (x=0.000000, y=0.000000)... 0.000000
```

C++ inheritance example

```cpp
  // Print objects using Tools class
  printf("\nprint objects ------------ \n");
  cout << "Tools::fPrint(a);" << endl;
  Tools::fPrint(a);
  a->Print();
  cout << "Tools::fPrint(buf[1]);" << endl;
  Tools::fPrint(buf[1]);
  buf[1]->Print();
  cout << "Tools::fPrint(c);" << endl;
  Tools::fPrint(c);
  c->Print();
  cout << "Tools::fPrint(a1);" << endl;
  Tools::fPrint(a1);
  cout << "Tools::fPrint(b1);" << endl;
  Tools::fPrint(b1);

  // delete all heap objects
  printf("\ndelete all objects ------------ \n");
  for (int i=0; i<N; i++) {
    cout << "delete buf[" <<i << "]" << endl;
    delete buf[i];
  }
  delete [] buf;
}
```

```
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void A::Print()] (x=3.141500)
[virtual void A::Print()] (x=3.141500)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void B::Print()] (x=4.220000, y=5.100000)
[virtual void B::Print()] (x=4.220000, y=5.100000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=9.800000, y=-2.000000)
[virtual void C::Print()] (x=9.800000, y=-2.000000)
[static void Tools::fPrint(A *)] typeid(p) = P1A
[virtual void C::Print()] (x=1.657000, y=-1.000000)
[virtual void C::Print()] (x=1.657000, y=-1.000000)

delete buf[0]
[virtual A::~A()] Destructor called...
delete buf[1]
[virtual B::~B()] Destructor called...
[virtual A::~A()] Destructor called...
delete buf[2]
[virtual C::~C()] Destructor called...
[virtual B::~B()] Destructor called...
[virtual A::~A()] Destructor called...
delete buf[3]
[virtual C::~C()] Destructor called...
[virtual B::~B()] Destructor called...
[virtual A::~A()] Destructor called...
delete buf[4]
[virtual C::~C()] Destructor called...
[virtual B::~B()] Destructor called...
[virtual A::~A()] Destructor called...
```

C++ class static members

- ✔ *class static member* is a member of a class and not of the objects of the class.
  there will be exactly one copy of the static member per class
- ✔ a function that needs to have access to members of a class but need not to be invoked for a particular object, is called a *static member function*

```
class vector2D {
  private:
   pointer2D *P;
   static point2D InitPoint; //init point defined for all objects
  public:
   static void SetInitPoint(const point2D& );
};
```

Note: private static data members cannot be accessed publicly (only from class members)

- ✔ Initializing static variable (in vector2D.C)

```
//init static variable with (0,0)
  point2D vector2D::InitPoint=point2D();
// implementation of the class code
  vector2D::vector2D(point2D p2, point2D p1) {
  ...
```

C++ class static members (cont.)

- ✔ Implementing static method (in vector2D.C)

```
//init static variable with (0,0)
  point2D vector2D::InitPoint=point2D();
// implementation of the class code (keyword static not needed)
  void vector2D::SetInitPoint(const point2D& ) {
    ...
  }
```

- ✔ calling static function from main.C

```
#include "vector2D.h"
#include "point2D.h"

  int main() {
    //call static function and set static variable to (1,1)
    vector2D::SetInitPoint(point2D(1,1));
  }
```

C++ class static methods

- ✔ *Static methods* can be implemented in classes in order to provide functions within a class scope that does not need any private member the class works as a repository of functions that have to be called from the user-function with the scope operator

```
class USERtools {
  public:
    // computes maximum value of array
    static double MaxValue(double*);
};

#include "USERtools.h"
int main() {
  double p[] = {0.23, 053, 2.3, 5.6, 7.};
  double result = USERtools::MaxValue(p);
}
```

C++ namespaces

- ✔ Names in C++ can refer to variables, functions, structures, enumerations, classes and class and structure members. The potential for name conflicts increases when number of code lines become big!

  The usual way to solve this problem in C language was to define some prefix common to all variables belonging to a same package!

- ✔ The C++ provides **namespace** facilities to provide greater control over the scope of names.

  The names in one namespace do not conflict with the same names declared in other namespaces

- ✔ To access names declared in a given namespace we can use the **scope-resolution operator (::)**

C++ namespaces (cont.)

particle.h (FCOMP namespace)

```
#ifndef __MYH__
#define __MYH__
#include <string>
namespace FCOM {
 //user function
 int addnumbers(int,int);
 int MyFlag=0;

 class particle {
   public:
     particle() {}; //default constr
     ...
     void SetMass(double);
   private:
     std::string name;
     double mass;
     int charge;
     ...
 }; //end of class
} //end of namespace
#endif
```

(particle.C) FCOMP namespace

```
#include "particle.h"
namespace FCOMP {
void particle::SetMass(double m) {
 mass = m;
}
}
```

user program

```
#include "particle.h"
...
int main() {
  //set variable value to 101
  FCOMP::MyFlag = 101;
  //call function
  int a = FCOMP::addnumbers(3,10);
  //instantiate object
  FCOMP::particle P;
  P.SetMass(0.511E-3);
}
```

C++ namespaces (cont.)

✔ The **using** declaration simplifies the procedure for using the names declared under a given namespace

```
using FCOMP::MyFlag;
```

after this declaration we can use directly the name **MyFlag**.
If we redeclare a variable **MyFlag** now, an error is returned!

✔ The **using namespace** declaration makes all names defined within the namespace directly accessible!
We do not need anymore the scope-operator to acess them!

```
// all names from FCOMP usable
using namespace FCOMP;
// all names from std usable
using namespace std;
```

Computational Physics

ROOT

A data analysis graphics tool with a C++ interpreter

Fernando Barao, Phys Department IST (Lisbon)

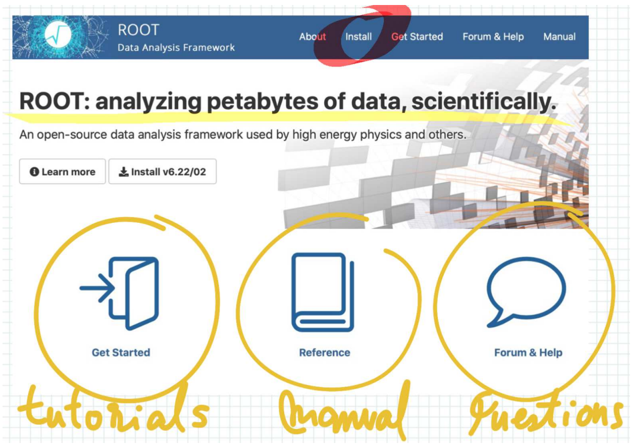
ROOT - outline

- ✔ ROOT installation
- ✔ general concepts
- ✔ interactive use and macros
- ✔ canvas and graphics style
- ✔ histograms and other objects
- ✔ fitting
- ✔ input/ouput
- ✔ using ROOT from user programs

site:	`http://root.cern.ch`
Users Guide:	`http://root.cern.ch/drupal/content/root-users-guide-600`

ROOT - introduction

- ✔ ROOT is and object oriented framework designed for solving data handling issues in High Energy Physics such as data storage and data analysis (display, statistics, ...)
- ✔ ROOT was the next step after the PAW data analysis tool developped in Fortran on $90's$ and widely used by physicists
- ✔ ROOT is supported by the CERN organization and it is continuously evolving
- ✔ ROOT is nowadays used in other fields like medicine, finance, astrophysics, ... as a data handling tool



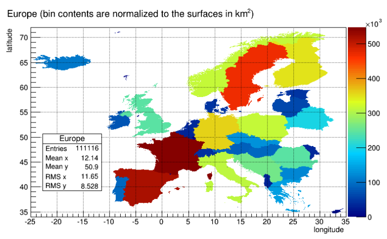
ROOT: analyzing petabytes of data, scientifically.

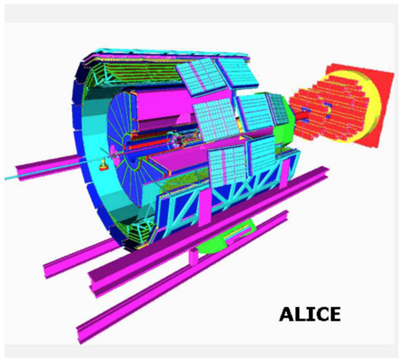
An open-source data analysis framework used by high energy physics and others.

ROOT - categories

many fields/categories covered:

- ✔ base: low level building blocks (TObject,...)
- ✔ container: arrays, lists, trees, maps, ...
- ✔ physics: 2D-vectors, 3D-vectors. Lorentz vector, Lorentz Rotation, N-body phase space generator
- ✔ matrix: general matrices and vectors
- ✔ histograms: 1D,2D and 3D histograms
- ✔ minimization: MINUIT interface,...
- ✔ tree and ntuple: information storage
- ✔ 2D graphics: lines, shapes (rectangles, circles,...), pads, canvases
- ✔ 3D graphics: 3D-polylines, 3D shapes (box, cone,...)
- ✔ detector geometry: monte-carlo simulation and particle tracing
- ✔ graphics user interface (GUI):
- ✔ networking: buttons, menus,...
- ✔ database: MySQL,...
- ✔ documentation





45-1

45-2

Computational Physics

ROOT

A data analysis graphics tool with a C++ interpreter

Fernando Barao, Phys Department IST (Lisbon)