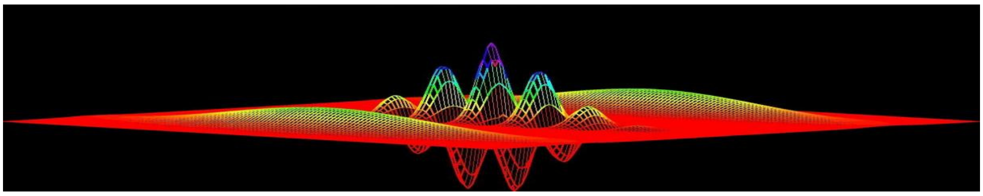
Computational Physics

numerical methods with C++ (and UNIX)



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: barao@lip.pt

class vector2D

Let's make a class **vector2D** making use of the class **point2D** before defined ; it will include two point2D data members dynamically allocated that will require the user to define copy's constructor and assignment

a possible class definition with two points (vector2D.h)

```
class vector2D : public cFC {
  public:
   vector2D(point2D pf, point2D pi) : cFC("A01", "2014-15"), Pf(pf), Pi(pi) { cout <<
   vector2D(point2D pf) : cFC("A01", "2014-15"), Pf(pf), Pi() { cout << "point2D const
  private:
   point2D Pi; //initial point
   point2D Pf; //final
};
```

class definition with a point2D pointer (vector2D.h)

```
class vector2D {
  public:
   vector2D(point2D pf, point2D pi);
   vector2D(point2D pf);
  private:
   point2D *P; //pointer
};
```

class implementation (vector2D.C)

```
vector2D::vector2D(point2D p2, point2D p1) {
  P = new point2D[2];
  P[0] = p1; P[1] = p2; }
vector2D::vector2D(point2D pf) {
  P = new point2D[2];
  P[0] = point2D(); //default constr
  P[1] = pf; }
```

class vector2D (cont.)

vector2D class : copy and assignment constructor declarations (vector2D.h)

```
class vector2D {
  public:
   vector2D(const vector2D&); //copy constructor
   vector2D& operator=(const vector2D&); //copy assignment
   ...
};
```

vector2D class : copy and assignment constructor implementation (vector2D.C)

```
   vector2D::vector2D(const vector2D& t) {//copy constructor
    P = new point2D[2]; // array with two points created
    P[0] = t.P[0];
    P[1] = t.P[1];
   }
   vector2D& vector2D::operator=(const vector2D& t) {//copy assignment
    if (this != &t) { //this is a const pointer to current object (member func invoked
       P[0] = t.P[0];
       P[1] = t.P[1];
    }
    return *this;
   }
```


class inheritance : virtual destructors

- ✔ A 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();
};

class Derived: public Base {
  public:
  ~Derived();
};
```

C++ classes polymorphism

- **pointers to base class**
  The class inheritance allows the polymorphic characteristic that a pointer to a derived class is type-compatible with a pointer to its base class
  A class method argument can use generically a pointer to the base class for passing any derived class object (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;
```

- **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 instatiated by itself !

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


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.)

✔ 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
  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 proble 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;
   //user class
   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;
```


speed up you program...

- It is important to realise that multiplication (*) and division operations (/)consume considerably more CPU time than addition (+), subtraction (-), comparison or assignment operations

➜ **try to avoid redundant multiplication and division operations**

- For instance try to define a 3rd-order polynomial written like this :

$$f(x) = P_0 + P_1\ x + P_2\ x^2 + P_3\ x^3$$

In total, we have 6 multiplication operations. We can optimize the polynomial expression to reduce the number of multiplications :

$$f(x) = P_0 + x\ \ (P_1 + x\ \ (P_2 + +P_3\ x))$$

The number of multiplications is now 3.

- math library tend to be extremely expensive in terms of CPU time

➜ **only use when absolutely necessary**
For instance, instead of $pow(x, 2)$ use $x * x$

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
- ✔ DUBNA

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 medecine, finance, astrophysics, ... as a data handling tool

ROOT - installation

- Download a source copy from ROOT svn server and install it at ~<user>/SOFT/root

```
# check which versions are there...
> svn ls https://root.cern.ch/svn/root/tags
# choose a recent version (tag) as it is v5-34-06 and check out
> mkdir SOFT; cd SOFT
> svn co https://root.cern.ch/svn/root/tags/v5-34-06 root-53406
# make a symbolic link (dir name aliases)
> ln -sf root-53406 root
> cd root
# define ROOT environment variable
> setenv ROOTSYS ~<user>/SOFT/root
> cd $ROOTSYS
# produce the makefile
> ./configure
> make distclean
> make
> make install
# Every time you run root
> source $ROOTSYS/bin/thisroot.csh #if you use cshell
```

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

ROOT - TH1 class inheritance