COMPUTATIONAL PHYSICS

Classes in C++

Static members

Inheritance and friendship

Polymorphism

Recalling Static variables

- We know static variables are instantiated only once and survive scope...

```
void foo(string name)
{
  static int n = 0; //unique identifier generator (-:
  n++; cout << n << endl; cout << "Greetings " <<
name << endl;
}
int main() {
  char tmp[21]; string name="";
  while (name != "chega") {
    scanf("%20s",&tmp);
    name=tmp;
    foo(name);
  }
return 0;}
```

```
Rui
1
Greetings Rui
Luis
2
Greetings Luis
Mariana
3
Greetings Mariana
Jose
4
Greetings Jose
chega
5
Greetings chega
```

- For Classes, static members can also be set but C++ forbids in-class initialization of non-const static member ! → ***do it outside*** !

Classes - Static member variables

- Static class members are ***not associated*** to class objects but to the class itself !!!

```
class Asteroid {
private:
  static int numOfAsteroids; //non-const
public:
  static int numSights;
  void setnA(int a) {numOfAsteroids=a;}
  int getnA() {return numOfAsteroids;}
 Asteroid() {numOfAsteroids++;}
 ~Asteroid(){numOfAsteroids--;}
};
int Asteroid::numOfAsteroids = 0;
int Asteroid::numSights = 0;
```

```
int main () {
  Asteroid A; //numOfAsteroids=1
  Asteroid::numSights=3;
  Asteroid B; //numOfAsteroids=2
  // FAIL numOfAsteroids is a private member
  //Asteroid::numOfAsteroids=12;
  A.setnA(12); // numOfAsteroids=12
  cout << "#A = " << A.getnA() << endl; //12
  cout << "#B = " << B.getnA() << endl; //12
  cout << "#sights = " << B.numSights; //3
return 0;
}
```

- If a static member variable is public we don't even need an object to set it's value !!! (*To ckeck: retain only second line in main above !*)

Classes - Static member functions

- Static class function can ***only*** access static member variables.
- They are ***not associated*** to class objects but to the class itself !

```
class Asteroid {
private:
  static int numOfAsteroids;
public:
  static int getnA() {return numOfAsteroids;}
  void setnA(int a) {numOfAsteroids=a;}
  Asteroid() {numOfAsteroids++;}
  ~Asteroid(){numOfAsteroids--;}
};
int Asteroid::numOfAsteroids = 0;
int main () {
  Asteroid A; //numOfAsteroids=1
  A.setnA(12); // numOfAsteroids=12
  cout << "#Ast. = " << Asteroid::getnA(); //12
return 0;
}
```

Static member function have no ***this** pointer....since there is no *dedicated* "this" object calling the function !

→ We cannot use ***setnA*** without a "calling" object !

→ Only static functions can return numOfAsteroids !

Classes – Relationships

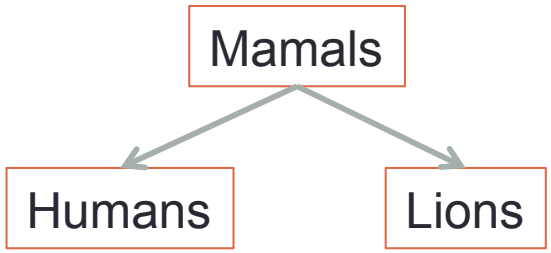
- Two objects may have many different kinds of relationships in different use cases e.g. “part-of”, “has-a”, ”uses-a” or “is-a”.
- Most commonly, we have
  - Classes with members that are objects of another class e.g. a ***Car Class*** with members *brand*, *power*, *seats* and ***position***. Position can be of type ***Location Class***.
  - Different classes with clear association between each other e.g. ***FiniteElement*** Class with a *vector of particles* member and a ***Particle Class*** that, among the members, has the *FiniteElement* it occupies.
  - Classes that are clearly *derived* from more fundamental classes e.g. ***Rectangle***, ***Square*** are clearly an “descendant” of basic P***olygon***

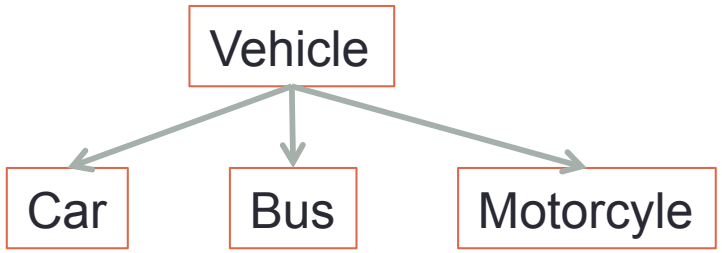
Let’s check the latter case → ***INHERITANCE***

Classes – Inheritance

- In a nutshell as ***basic principle***: when different classes *share* many common data *members* and *methods* even if with different implementations → use ***Inheritance***

Examples





- All mamals have weight, age and body temperature.
- But Lions don't buy shoes...
- ...and Humans don't wag a tail...

- All vehicles have an engine and weels and we can *apply breaks*
- But we can't remove the hard top of a motorcyle !
- ...nor make a wheely with a bus...

Classes – Inheritance: access

- The inheritance relationship of two classes is *declared* in the derived class

→ Access to *base class* by *derived class* is regulated by an ***access specifier***

```
class Polygon {
...member variables / methods...
};
class Rectangle: public Polygon {
  private:
    .....
  public:
    .....
};
```

Rule of thumb → Inheritance access sets minimum privilege on base class members !

Public inheritance:

public (base) → public (derived)

protected (base) → protected (derived)

private (base) → not accessible (derived)

Private inheritance:

public (base) → private (derived)

protected (base) → private (derived)

private (base) → not accessible (derived)

Only public or protected members from base class are accessible directly in derived class (even with public inheritance) !

Classes – Inheritance : exceptions

- In inheritance, not every member is inherited (directly accessible)

  => its constructors and its destructor
  => its assignment operator members (operator=)
  => its friends
  => its private members

- Though not inherited, the base class constructor and destructor is ***ALWAYS*** called.

```
class Parent {
...member variables / methods...
};
class Child: public Parent {
  private:   .....
  public:    .....
};
Int main(){
  Child Son; // Parent const.called
  Child Daughter; //idem
} //2 Parent Destructor called.
```

- In case a Parent constructor exists, the Child constructor can (should) call it !

```
Child Patrick(int a): Parent(a) {.....};
```

✧ If only we could inherit base class methods as "*templates*" to freely implement as required by each derived classes…

Classes – Inheritance : virtual functions

- We can always *redefine* a base class function on each derived class. But we should ***NOT*** do it…it will fail miserably when defining base class references/pointers to derived objects (*C++ allows this...*) !

```
class Human {
 protected:
   int weight, height;
 public:
   void salute () { cout << "???" << endl; }
};
class German: public Human {
 public:
   German(int a=95,int b=188)
{weight=a; height=b;}
   void salute() { cout << "Guten
Morgen !" <<  endl;}
   string set_ID(int x, string y) {...code...}
};
```

```
int main () {
  German Hans;
  Hans.salute(); //Guten Morgen !
  Human * MrA=&Hans;
  MrA->salute(); // guess what ?
  Human & MrB=Hans;
  MrB.salute(); // guess what ?
  MrB.set_ID(123,"kjaz"); // possible ?!
```

Virtual keyword to the rescue !

```
class Human {
  public:
   virtual void salute () { cout << "???" << endl; }
   virtual set_ID(int x, string y) {}
};
```

Classes – Inheritance : virtual functions

```
class Human {
  protected:
    int weight, height;
  public:
   virtual  void salute () { cout << "???" <<
endl; } };
class German: public Human {
  public:
    German(int a=95,int b=188)
{weight=a; height=b;}
    void salute() { cout << "Guten
Morgen !" <<  endl;} };
class Swedish: public Human {
  public:
    Swedish(int a=70,int b=175)
{weight=a; height=b;}
    void salute() { cout << "God Morgon !"
<<  endl;} };
```

```
int main () {
  German Hans;
  Hans.salute(); // Guten Morgen !
  Human * MrA=&Hans;
  MrA->salute(); // Guten Morgen !
  Swedish Sven;
  Human & MrB=Sven;
  MrB.salute(); // God Morgon !
}
```

Very useful: we may define an array of type “Human” with *different class objects* and use the *same* method call !

Classes – Inheritance : abstract classes

- If a virtual function in a base class is “undefined” e.g. is unclear meaning, one can always not define it at all ! ➡ *Pure virtual function*
- A Class with at least one pure virtual function is called ***Abstract Class***.
- If one doesn’t overload a pure virtual function in a derived class, it becomes also an abstract class !
- This is all so relevant since **NO OBJECTS OF AN ABSTRACT CLASS CAN BE INSTANTIATED** ! …the class becomes an “interface” type object...

```
class Human {
  protected:
    int weight, height;
  public:
    virtual  void salute () = 0
    virtual string get_ID() = 0;
    void print_ID() { cout <<
        this->get_ID() << endl;}   };
```

```
class German: public Human {
  public:
    string ID;
      ....
    string get_ID() {return ID;}
};
```

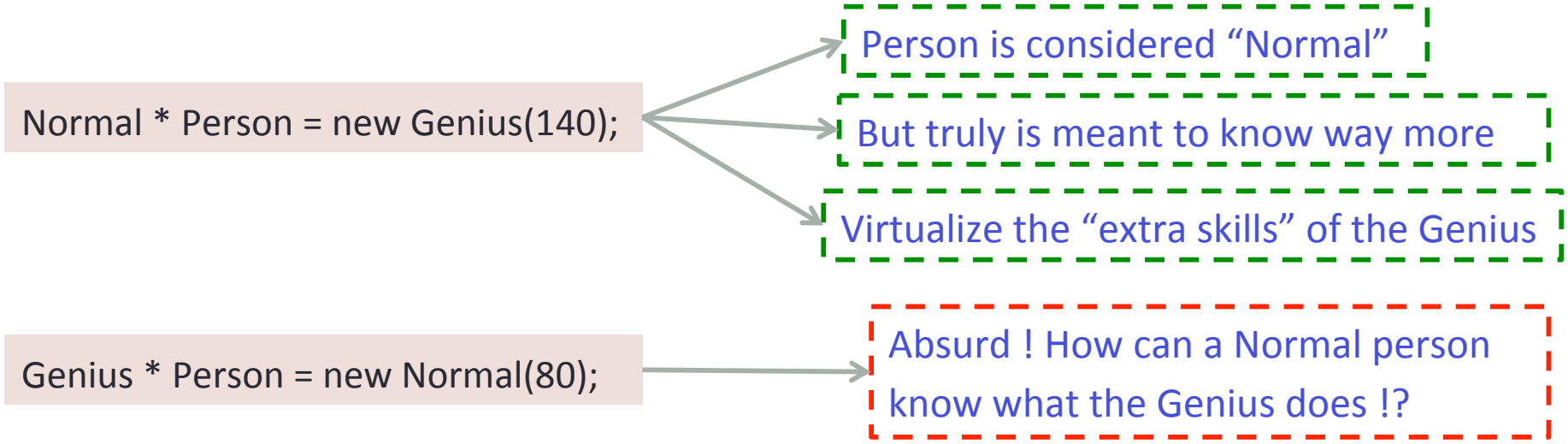
Abstract base classes may use the ***this*** pointer !

Classes – Inheritance : remarks

- Inheritance is a technique to build new Classes from old classes.
- The *derived* class is meant to be used much like the *base* class since it *inherits* (with some access rights) the data members and methods.
- The derived class can extend the *portfolio* with new functions or variables → not accessible to base class objects !
- It is perfectly possible to define **base class pointers or references** to **derived class objects** ! Virtualization in base class methods is ***KEY***.
- As base class constructor & destructor is ***ALWAYS*** called, ***virtualizing base class destructor*** ensures base class pointer to derived class is properly cleaned up !
- Static variables in base class are “propagated” to every derived class → derived class or base class pointers objects readily access them.

Classes – Polymorphism

- It is perfectly possible to define **base class pointers or references** to **derived class objects** ! Virtualization in base class methods is ***KEY***.
- A class that declares or inherits a virtual function is called a ***polymorphic class***..




```
Dynamic casting: Genius * Happy = dynamic_cast<Genius *>(Person);
```

***NOTE*: a bad dynamic cast is only found at runtime → if (Happy) gives false**

Classes – Inheritance : Full example

Diagnostics.h

```
#ifndef Diagnostics_H       //INCLUDE SAFEGUARD
#define Diagnostics_H       //INCLUDE SAFEGUARD
#include <iostream>
#include <vector>
using namespace std;

class Diagnostics {
protected:
   string name;
   float price;
   vector<double> time,data;
public:
  Diagnostics(string n="",float p=0.0);
  virtual ~Diagnostics();
  void set_time(vector<double> t);
  void set_data(vector<double> sig);

  void print_signal();

  virtual void set_wavelength (double);
  virtual void set_area (double);
  virtual vector<double> get_temperature ();
  virtual vector<double> get_field ();
};
#endif                      //INCLUDE SAFEGUARD
```

Diagnostics.cpp

```
#include <iostream>
#include <vector>
#include "Diagnostics.h"
using namespace std;

Diagnostics::Diagnostics(string n,float p) {name=n, price=p;
            cout << __PRETTY_FUNCTION__ << endl;}
Diagnostics::~Diagnostics() {cout << __PRETTY_FUNCTION__
<< endl;}

void Diagnostics::set_time(vector<double> t) {time=t;}
void Diagnostics::set_data(vector<double> sig) {data=sig;}

void Diagnostics::print_signal() {
  int np=time.size();
  printf("Data in diagnostic %s\n",name.c_str());
  for (int i=0;i<np;i++)
    printf("%f  %f\n",time[i],data[i]);
  cout << endl;
}
void Diagnostics::set_wavelength (double) {};
void Diagnostics::set_area (double) {};
vector<double> Diagnostics::get_temperature () {};
vector<double> Diagnostics::get_field () {};
};
```

Classes – Inheritance : Full example

Coil.h

```
#ifndef Coil_H
#define Coil_H
#include <iostream>
#include <vector>
#include "Diagnostics.h"
using namespace std;

class Coil: public Diagnostics {
public:
  double area;
  vector<double> field;
  Coil(string a,float b,double c);
  ~Coil();
  vector<double> get_field();
  void set_area(double a);
};
#endif
```

Coil.cpp

```
#include <iostream>
#include <vector>
#include "Diagnostics.h" //not actually needed !
#include "Coil.h"
using namespace std;

Coil::Coil(string a,float b,double c): Diagnostics(a,b) {area=c;
cout << __PRETTY_FUNCTION__ << endl;}
Coil::~Coil() {cout << __PRETTY_FUNCTION__ << endl;}
vector<double> Coil::get_field() {
  int np=time.size();
  for (int i=0;i<np;i++)
    field.push_back(data[i]/area);
  return field;
}
void Coil::set_area(double a) {area=a;}
```

→ The include safeguard avoids the duplication of the Header file when including it !

Classes – Inheritance : Full example

Laser.h

```
#ifndef Laser_H
#define Laser_H
#include <iostream>
#include <vector>
#include "Diagnostics.h"
using namespace std;

class Laser: public Diagnostics {
public:
  double wavelength;
  vector<double> temperature;
  Laser(string a,float b,double c);
  ~Laser();
  vector<double> get_temperature();
  void set_wavelength(double a);
};
#endif
```

Laser.cpp

```
#include <iostream>
#include <vector>
#include "Diagnostics.h" //not actually needed !
#include "Laser.h"
using namespace std;

Laser::Laser(string a,float b,double c): Diagnostics(a,b)
{wavelength=c; cout << __PRETTY_FUNCTION__ << endl;}
Laser::~Laser() {cout << __PRETTY_FUNCTION__ << endl;}
vector<double> Laser::get_temperature() {
  int np=time.size();
  for (int i=0;i<np;i++)
    temperature.push_back(data[i]/wavelength);
  return temperature;
}
void Laser::set_wavelength(double a) {wavelength=a;}
```

Classes – Inheritance : Full example

main.cpp

```
#include <iostream>
#include <vector>
#include "Diagnostics.h" //not needed since Coil/Laser include it !
#include "Coil.h"
#include "Laser.h"
using namespace std;

int main () {
  Diagnostics * MAGN=new Coil("Mirnov",15.0,0.01);
  Diagnostics * HRTS=new Laser("Thomson",17500.0,660.0e-9);
  MAGN->set_area(0.01);
  int nt=10;
  vector<double> time(nt),data(nt);
  for (int i=0;i<nt;i++) {
    time[i]=(double)i*0.01; data[i]=( (double)i*0.01 )*( (double)i*0.01 );
  }
  MAGN->set_time(time); MAGN->set_data(data);

  for (int i=0;i<nt;i++) {
    time[i]=(double)i*0.001; data[i]= (double)i*0.5 + 25.0;
  }
  HRTS->set_time(time); HRTS->set_data(data); cout<<endl;
  MAGN->print_signal(); HRTS->print_signal();
  delete MAGN; delete HRTS;
}
```

To compile:

g++ *.cpp -o main.exe