COMPUTATIONAL PHYSICS

Classes in C++

Overall concept: object, members, methods

Constructor, initialization, overloading

Copying, assigning, moving and destructing

Operators overloading

Classes and object oriented programming

- C++ brings object orientation to the C programming language !
- ***Classes*** are central for C++ concept of object oriented programming → classes are often depicted as user-defined types/structures
  - A user defined ***Name***
  - ***Data members*** – data variables defined inside the class
  - ***Member functions*** – functions used to "manipulate"/query the data members.
- Every instantiation of a class is called an ***Object*** e.g. *Car mycar*;
- To define a Class :

the Class **the Object**

```
class Car {
  access_specifier:
    data member1;
    function member1;
};
```

public
protected,
private

Classes and objects

```
class Car {
  access_specifier:
    data member1;
    function member1;
};
```

public **-** members can be accessed from anywhere the object is visible
private **–** members can be accessed only from class members
protected **–** same as private but access also granted from "derived classes" (see ***Inheritance***)

- By default, class members have private access → specifying private is not mandatory but advisable (*code readability*)

```
class Car {
    int seats;
    double power;
  public:
    string brand; //not really OO programming !
    int Get_seats();
    void Set_power(double);
};
```

```
int Car::Get_seats() {
  return seats;
}
void Car::Set_power(double pwr) {
  power=pwr;
}
```

Classes and Structures

- Structure data type is similar to a Class but without the access specifier for the data members → they are all public !

```
structure Car {
  int seats;
  double power;
  string brand;
  int Get_seats() {
    return seats;};
  void Set_power(double);
};
void Car::Set_power(double
pwr) {power=pwr;}
```

```
class Car {
    int seats;
    double power;
  public:
    string brand;
    int Get_seats() {return seats;};
    void Set_power(double);
};
void Car::Set_power(double pwr)
{power=pwr;}
```

```
Int main {
Car XP; ; //seats=?, power=?,...
XP.power=223.0; //don't need to
use Set_power !
}
```

```
Int main {
Car XP; //seats=?, power=?,...
XP.power=223.0; // FAIL to compile !
XP.Set_power(223.0);
}
```

Class constructor

- Declaring an object (variable) of a given Class doesn't automatically set member variables value unless we code it → ***Constructor*** member !

```
class Car {
    int seats;
    double power;
  public:
    int Get_seats() {return seats;}
    void Set_power(double)  {
      power=pwr;}
    Car(int,double);
};
Car::Car(int a, double b) {seats=a;
power=b;}
```

We can easily create a new object Car, setting *seats* and *power* !

Car myCar(5,65.0);

```
class Car {
    int seats;
    double power;
  public:
    int Get_seats() {return seats;}
    void Set_power(double pwr)  {
      power=pwr;}
    Car(int,double);
};
Car::Car() {seats=5; power=100.0;}
```

Once we create a new object Car, we can only set the power !

Car myCar; //seats permanently 5

→ ***Default constructor***

Class constructor - overloading

- When defining functions in C++, we saw we can "overload".

```
int SUM(int a, int b)
double SUM(double a, double b)
```

- For class member functions, the same happens

```
class Car {
    int seats;
    double power;
  public:
    int Get_seats() {return seats;}
    void Set_power(double pwr)  {
      power=pwr;}
    Car(); Car(int,double);
};
Car::Car() {seats=5; power=100.0;}
Car::Car(int a, double b) {seats=a;
power=b;}
```

Car.h

```
class Car {
    int seats; double power;
  public:
    int Get_seats(); void Set_power(double);
    Car(); Car(int,double);
};
```

Car.cpp

```
#include "Car.h"
Car::Car() {seats=5; power=100.0;}
Car::Car(int a, double b) {seats=a; power=b;}
int Car::Get_seats() {return seats;}
void Car::Set_power(double pwr)  {power=pwr;}
```

Class initialization – members and object

- There are multiple ways to initialize members and the object

```
class Aluno{
    string name;
    int number;
  public
    Aluno(string name=“Carlos”,int number=1);
};
Aluno::Aluno(string a, int b) {name=a; number=b;}
      or
Aluno::Aluno(string a, int b): name(a) {number=b;}
      or
Aluno::Aluno(string a, int b): name(a), number(b) {}
```

Iniitialization list

```
Aluno a1; Aluno2 a2; //default

Aluno b1("Rui",35492);
Aluno b2("Catarina"); //number=1
Aluno b3={"Pedro",41000};
Aluno b4=(36323); // FAIL
Aluno b5={”Helena”}; // FAIL
```

Classes – copying, assigning, moving

- Once we have an object of a given class, we frequently need to *construct* an object with content to another one or *assign* one object content to another one

- To be more precise, we might want to do a ***COPY*** or simply ***MOVE*** the data from one object to the other.

- Digging even more deeply, we might be smart enough (*a must...*) to ensure that any dynamic memory allocation is properly managed and we don't get e.g. *dandling pointers*...

- Let us then learn with some examples the fundamentals of
  - COPY constructor
  - COPY assignement
  - MOVE constructor
  - MOVE assignement

On the road we will learn about the this keyword

Classes – copy constructor

- Build an object from an existing object !
- So...how to call it ?

```
Aluno a1; //default constructor...
Aluno a2(a1);  //one way to copy
Aluno a3=a1; // another way to do it
```

- And how to implement it ?

```
class Aluno{
    string name;
    int number;
  public:
    Aluno(const Aluno &al) {
      name=al.name;
      number=al.number;
}
```

- If we have pointers, ***allocate*** !

```
class Aluno{
    string name; int * number;
  public:
    Aluno(const Aluno &al) {
      name=al.name; number=new int;
      *number=al.number;
}
```

- We can use *References* to get the object without copying it !

```
Aluno & const getAluno() {
    return *this; //this one calling you...
}
Aluno a1(“Luis”,94694);
Aluno &a2=a1; //no copy constructor
Aluno a2=a1.getAluno(); //copy constructor
```

Classes – copy assignement

- We first declare the object and then assign to it a copy of another existing one.

Syntax

```
Aluno a1("Ema",89678);
Aluno a2;
a2=a1; // assign a copy of a1 to a2
```

Implementation

```
class Aluno{
    string name; int * number;
  public:
    Aluno & operator=(const Aluno & al) {
      if (this != &al) {
        name=al.name;
        number=new int;
        *number=*al.number;}
      return *this;}
}
```

Deep copy

- Whenever we have pointers we NEED to allocate memory for new pointer → ***deep*** vs ***shallow*** copy

```
Aluno & operator=(const Aluno & al) {
      name=al.name;
      number=al.number;  // Shallow copy
      return *this;
}
```

With shallow copy you get a dangling pointer after destroying the source object...

Classes – move constructor

- ***Aim***: construct an object *moving* content from a *temporary* object → source object is unnamed

Syntax

```
Aluno a1("Ema",89678);
Aluno a2=Aluno("Tim",67877);
```

Implementation

```
class Aluno{
     string name; int * number;
  public:
     Aluno(Aluno &&al) {
       name=al.name;
       number=al.number;
       al.number=nullptr; //MANDATORY !
     }
}
```

- After taking the content of the source, nullify any pointer member !

Why ?

- On exit of constructor, temporary is destroyed → *dangling pointer*

N.B. C++ compilers are "smart" and use *Return Value Optimization*. To force use of ***move*** constructor use flag *-fno-elide-constructors* in compilation.

Classes – move assignement

- ***Aim***: assign an object *moving* content from a *temporary* object → source object is unnamed

Syntax

```
Aluno a1("Ema",89678);
Aluno a2;
a2=Aluno("Tim",67877);
```

Implementation

```
class Aluno{
    string name; int * number;
  public:
    Aluno & operator=(Aluno &&al) {
      delete number; //release memory
      name=al.name;
      number=al.number; //copy content
      al.number=nullptr; //MANDATORY !
      return *this;
    }
}
```

- If taking possession of source content → release first already allocated memory !

Classes – destructor

- ***Aim***: when an object is no longer needed and we want to release/free the occupied memory (*or exit the execution...*)
- It takes no arguments and returns nothing, uses class name in it's name, preceeded by a ~.
- It need **NOT** be called: automatically if *temporary* or when *exiting scope* !

Implementation

```
class Aluno{
    string name; int * number;
  public:
    Aluno(string a="Carlos",int b=1): name(a),
number(new int(b)) {}
    ~Aluno() {
      delete number;
    }
```

Use cases

```
Int main (){
  Aluno a1("Ema",89678); //constructor
  Aluno a2=Aluno("Tim",67877);
//construct temporary → move construct
and finally destruct temporary !
}
//exiting scope, destruct a1 and a2 !
```

Classes – special member functions

- If we don't provide dedicated member function for constructor, destructor,... some implicit definitions are assumed !

Default constructor	Cl::Cl()	User provided constructor lacking
Destructor	Cl::~Cl()	Lacking a destructor
Copy constructor	Cl::Cl(const Cl &)	Lacking a move constructor or assignement
Copy assignement	Cl & operator=(const Cl &)	Lacking a move constructor or assignement
Move constructor	Cl::Cl(Cl &&)	Lacking destructor, copy constructor and also copy or move assignment
Move assignement	Cl & operator=(Cl &&)	Lacking destructor, copy constructor and also copy or move assignment

Classes – operators overloading

- The C++ is not obliged to know what typical operators e.g. Arithmetic mean when dealing with classes !
- Luckily, it allows us nonetheless to *overload* many operators for each Class definition !

=	Assignement operator
+, - , *	Arithmetic operators
+=, -=, *=	Compound arithmetic operators
==, !=	Comparison operators
++, - -, - !	Unary operators

- Operator overload done with regular functions with special names: their name begins by the ***operator*** keyword followed by the operator ***sign*** that is overloaded.

type operator sign (parameters) { /*... body ...*/ }

Classes – operators overloading options

- Where/How do i define the “overload operator” function ? Three options actually exist !

By member function

```
class ACME{
    private members
  public:
    ACME(...) {...} //constructor
    ACME operatorXYZ(ACME obj, parameters) {
      ...body...;
      return Object;}
}
int main {
ACME Obj1, Obj2;
ACME Obj3=Obj1.operatorXYZ(Obj2,parameters);
//Example: Obj3=Obj1+Obj2;
}
```

- The member function has direct access to the private members of the class.
- The function IS a member function of the class → must be called from an object of that class
- Implemented outside class definition:

```
ACME ACME::operatorXYZ(ACME obj,
parameters) { ...body...;
      return Object;}
```

Classes – operators overloading options

By friend function

```
class ACME{
    private members
  public:
    ACME(...) {...} //constructor
    friend ACME operatorXYZ(ACME Obj1, ACME Obj2);
}
ACME operatorXYZ(ACME Obj1, ACME Obj2) {
...direct access to ACME member variables...
return Object}

int main {
ACME Obj1, Obj2;
ACME Obj3=operatorXYZ(Obj1,Obj2);
//Example: Obj3=Obj1+Obj2; //if no operator+ member
//              Obj3=operator+(Obj1,Obj2); //force use
}
```

- The function is *allowed* direct access to the private members of the class.
- But we CANNOT use the *this keyword to access data members !
- The function IS NOT a member function of the class → requires 2 objects
- CAVEAT: if an equivalent member function exists it has precedence !
- TRIVIA: is any of this useful for TYPE “op.” OBJ instead of OBJ “op.” TYPE ?!

Classes – operators overloading options

By normal function

```
class ACME{
     private members
  public:
     ACME(...) {...} //constructor
}
ACME operatorXYZ(ACME Obj1, ACME Obj2) {
...must use Get() methods to fetch data members...
return Object}

int main {
ACME Obj1, Obj2;
ACME Obj3=Obj1.operatorXYZ(Ob2);
//Example: Obj3=Obj1+Obj2; //if no operator+ member
//              Obj3=operator+(Obj1,Obj2); //force use
}
```

- The function is NOT *allowed* direct access to the private members of the class.
- We need to use Get() methods first…
- The function IS NOT a member function of the class → requires 2 objects
- CAVEAT: if an equivalent member function exists it has precedence !

- Let's now see some examples…

Classes – operator overloading example 1

```
class Point {
  int x, y;
public:
  Point(int x = 0, int y = 0); // Constructor
  ~Point(); //Destructor
  int getY() const;
  int getX() const;
  void print() const;

  friend Point operator+(const Point & lhs,const Point & rhs);
  Point operator+(const Point & rhs);
  Point operator-(const Point & rhs);
  Point & operator=(const Point & rhs);
  Point & operator+=(const Point & rhs);
};
```

- Class declaration, friend function also declared

Classes – operator overloading example 1

```
// Getters
int Point::getY() const{return y; }
int Point::getX() const{return x; }
// Member Functions
void Point::print() const {
  cout << "(" << x << "," << y << ")" << endl;}
Point & Point::operator=(const Point & rhs) {
  if (&rhs != this){
    x=rhs.x; y=rhs.y; }
  return *this;}
Point Point::operator+(const Point & rhs) {
  return Point(x + rhs.x, y + rhs.y);}
Point Point::operator-(const Point & rhs) {
  return Point(x - rhs.x, y - rhs.y);}
Point & Point::operator+=(const Point & rhs) {
  x+=rhs.x;
  y+=rhs.y;
  return *this;}
```

```
//Friend function
Point operator+(const Point & lhs,const Point
& rhs) {
  return Point(lhs.x + rhs.x, lhs.y + rhs.y);}

//Normal function
Point operator-(const Point & lhs,const Point
& rhs) {
  int lx=lhs.getX(), ly=lhs.getY();
  int rx=rhs.getX(), ry=rhs.getY();
  return Point(lx - rx, ly - ry);}

int main() {
  Point p1(1, 2), p2(4, 5);
  Point p3=p1+p2; //member function if exist !
  Point p4=p1-p2; //likewise…
  p4+=p3;
  return 0;
}
```

Classes – operator overloading example 2

Object x constant

```
Point Point::operator*(int c) {
  return Point(c*x, c*y);
}
```

- Object (source unchanged)

```
Point Point::operator-() {
  return Point(-x, -y);
}
```

Object1 == Object2

```
bool Point::operator==(const Point & rhs) {
  bool res=(x==rhs.x && y==rhs.y);
  return res;
}
```

Main

```
int main() {
  Point p1(1, 2), p2(4, 5);
  Point p3=p1+p2;
  if (p1 == p3)
    cout << "The two points are the same !”;
  (p4*5).print();
  (-p4).print();
  return 0;
}
```

Classes – operators backstage…

```
int main() {
  Point p1(1, 2);
  Point p2(p1);
  Point p3=p1+p2;
  Point p4;
  p4=p1-p2;
  (p4*5).print();
return 0;
}
```

```
Point::Point(int, int)
```

```
Point::Point(const Point&)
```

```
Point Point::operator+(const Point&)
Point::Point(int, int)
Point::Point(Point&&)
Point::~Point()
Point::Point(Point&&)
Point::~Point()
```

```
Point::Point(int, int)
Point Point::operator-(const Point&)
Point::Point(int, int)
Point::Point(Point&&)
Point::~Point()
Point& Point::operator=(Point&&)
Point::~Point()
```

```
Point Point::operator*(int)
Point::Point(int, int)
Point::Point(Point&&)
Point::~Point()
(10,20)
Point::~Point()
```

Classes – operators backstage…

```
int main() {
  Point p1(1, 2);
  Point p2(p1);
  Point p3=p1+p2;
  Point p4;
  p4=p1-p2;
  (p4*5).print();
return 0;
}
```

Point Point::operator+(const Point&) → enter p1.operator+(p2)
Point::Point(int, int) → construct new object (p1+p2 content)
Point::Point(Point&&) → move construct it to a temporary-1
Point::~Point() → destruct the "p1+p2" object
Point::Point(Point&&) → move construct to a new temporary-2
Point::~Point() → destruct the temporary-1

Point::Point(int, int) → construct p4
Point Point::operator-(const Point&) → enter p1.operator-(p2)
Point::Point(int, int)) → construct new object (p1-p2 content)
Point::Point(Point&&) → move construct it to temporary-1
Point::~Point() → destruct the "p1-p2" object
Point& Point::operator=(Point&&) → move assign to p4
Point::~Point() → destruct the temporary-1

Point Point::operator*(int) → enter p4.operator*(5)
Point::Point(int, int) → construct new object (p4*5 content)
Point::Point(Point&&) → move construct to temporary-1
Point::~Point() → delete the p4*5 object
(10,20)
Point::~Point() → destruct temporary since not assigned after !