Class hierarchies

- The ***inheritance*** represents a distinctive element of the object-oriented programming paradigm
- Inheritance is also called a ***derivation*** operation: a class *A*, which inherits the members of another class *B*, it is said to be ***derived*** from the class *B*, and the class *A* is said a ***base class*** for *B*.
- In the UML language the inheritance relation is denoted as, $\uparrow$, where the arrow being oriented to the base class

Example. The classes *P* and *Q* inherit the class *X*.

$$\begin{tikzcd}
 & \boxed{X} & \\
\boxed{P} \arrow[ur] & & \boxed{Q} \arrow[ul]
\end{tikzcd}$$

```
class X {
  // ...
   };
class P: public X {
  // ...
};
```

```
class Q: X {
  // ...
};
```

Python example:

```
class X:
    pass

class P(X):
    pass

class Q(X):
    pass
```

A. Class inheritance

- A class can inherit only one class (***simple inheritance***), or several classes (***multiple inheritance***).
- A class hierarchy can be build based on the inheritance relation.
  - When using ***simple inheritance***, the class hierarchy has a *tree structure*, all classes from the hierarchy being derived from a single base class.
  - In the case when ***multiple inheritance*** is used, the hierarchy structure represents an *oriented graph*.

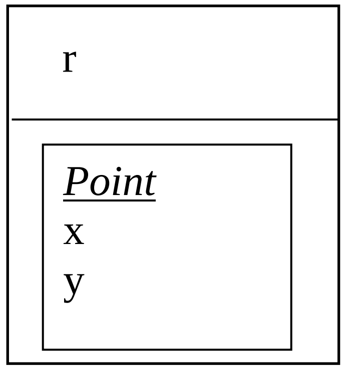
Remark. *Unions* cannot be part in a class hierarchy, because they *cannot* be *base classes*, neither *derived classes*.

Example.

```
struct Point {
  double x,y;
  vo id SetCoord (double a, double b) {
    x = a;
    y = b;
  }
};

struct Circle: Point {
  double r;
  void SetRadius (double a) { r = a; }
};
```

- ❑ An object from a derived class is treated by the compiler as a special type of composition relation.
- ❑ The memory image of an object of a derived class contains:
  - A *slot* for *each member* specified in the derived class,
  - An *additional slot* representing a ***hidden object***, which is an instance of the base class.

- For example, for the previous hierarchy (*Point* – *Circle*), the memory image of an instance of the class *Circle* has the following structure:



- A *base class* can be inherited in a *derived class* in three ways:
  - ***public***, ***protected***, and ***private***.
- In the case when this keyword is missing, the inheritance is by default ***public*** for *struct* and ***private*** for *class*.

- The access at the members of the base class can be sometimes restricted in the derived class, but never more restrictively:

The access type of a member in the base class	*Inheritance type*	*The access type of the member in the derived class*
Public Private Protected	Public	Public ***Inaccessible*** Protected
Public Private Protected	Private	*Private* ***Inaccessible*** *Private*
Public Private Protected	Protected	*Protected* ***Inaccessible*** Protected

- ***Python*** has only public inheritance

Example.

```
class A {
  int p;
public:
  int q;
protected:
  int r;
// ...
};

class A1: A {
  int x;
public:
  void f1();
// ...
};
```

```
class A2: public A {
  int y;
public:
  void f2();
// ...
};

void Processing() {
  A1 a1;
  A2 a2;
  // Error !!!
  int m = a1.q;
  // OK
  int n = a2.q;
}
```

- It is possible as certain public members from the base class, that become *private* by a inheritance of private type, to become *public* in the derived class:
  - their name must be specified after the keyword ***public*** (only the name, not its associated data type).

Example.

```
class A1: A {
  int x;
public:
  A::q;
  void f1();
  // ...
};
```

- ***Python*** has only *public inheritance*

- The ***memory structure*** of an instance in ***Python*** is totally different from that of the C++ language
  - A *class instance* has a *namespace* implemented as a *dictionary* that contains *instance attributes*
  - A *class* has also a *namespace* implemented as a *dictionary* that contains *class attributes*
  - First, the name of an attribute is searched in the dictionary of the current instance
  - If the name is not found, then it is searched in the dictionary of the class of the current class

- **Example**:

```
class A:
    k = 1
    def __init__(self, a):
        self.a = a

class B(A):
    def __init__(self, a, b):
        super().__init__(a)
        self.b = b

class C(A):
    def __init__(self, c):
        self.c = c
    def f(self):
        print(k)

a = A(7)
b = B(7, 5)
c = C(3)

print('a.dict = ', a.__dict__)
print('b.dict = ', b.__dict__)
print('c.dict = ', c.__dict__)
print('A.dict_keys = ', A.__dict__.keys())
print('C.dict_keys = ', B.__dict__.keys())
print('C.dict_keys = ', C.__dict__.keys())
```

The output:

```
a.dict =  {'a': 7}
b.dict =  {'a': 7, 'b': 5}
c.dict =  {'c': 3}
```

```
A.dict_keys =  dict_keys(['__dict__', '__weakref__', 'k',
   '__init__', '__doc__', '__module__'])
C.dict_keys =  dict_keys(['__module__', '__init__',
   '__doc__'])
C.dict_keys =  dict_keys(['__module__', 'f', '__init__',
  '__doc__'])
```

B. Constructors and destructors in the class hierarchies

- The ***constructors*** and ***destructors*** are the only functions of a base class (excepting the *assignment operator*) which ***cannot be inherited*** in a derived class:
  - Creation and destruction of these objects belong to their class and they cannot be passed down in a class hierarchy

- An object of derived class is similar to a composed object, which contains a hidden sub-object of the base class:
  - The *constructor of the object* of the derived class must *call first the constructor of the base class*

- The *call of the constructor* of the hidden sub-*object can be realized explicitly in the constructor initializer list* of the derived class (using the name of the base class)

Example.

```
class Point {
protected:
  double x, y;
public:
  Point(double a = 0, double b = 0) {
    SetCoord (a, b);
  }
  void SetCoord(double a, double b) { x = a; y = b; }
  double X() const { return x; }
  double Y() const { return y; }
};

class Circle: public Point {
  double r;
public:
  Circle(double a = 0, double b = 0, double c = 1):
    Point(a,b), r(c) { }
  Circle(Circle &c): Point(c.x, c.y), r(c.r) {}
  void SetR(double a) { r = 0; }
};
```

- A constructor for a class ***can be automatically generated*** by the compiler, in the case when the respective *class does not have a declared constructor*.

- There is ***an exception***, in the case when a class *is derived from one or more base classes*, and *all these base classes have explicit constructors which contain parameters*:
  - In this case *the derived class must contain an explicit constructor* that explicit *calls the constructors of the base classes that have parameters*.

- The ***destructors*** of the objects of the derived classes call by default the destructors of the objects of the derived classes *in reverse order* of the constructors call order.
- For example, for an object *ob* of a class *D* derived from a class *B*, the order of calling destructors is:
  - destructor of the *ob* object (in the class *D*)
  - destructor of class *B*
  - destructors of supplementary members of class *D*
  - destructors of members from the class *B*
- In the case of *multiple inheritance*, the *order of calling the constructors* of the objects from the base classes *is the order of their declaration* in the derived class
  - the calling of *destructors* is performed *in reverse order*

Example. A macrodefinition is used for defining classes.

```
#include <iostream>
using namespace std;

#define CLASS(ID) class ID {\
  public:\
    ID(int){cout<<"Class Constructor  "<<#ID<<endl;}\
    ~ID(){cout<<"Class Destructor  "<<#ID<<endl;}\
  };

CLASS(B1);
CLASS(B2);
CLASS(M1);
CLASS(M2);

class D: public B1, B2 {
  M1 m1;
  M2 m2;
public:
  D(int): m1(10), m2(20), B1(30), B2(40) {
    cout << "Class Constructor D" << endl;
  }
  ~D() { cout << "Class Destructor D" << endl; }
};

int main() {
  D d(0);
  // ...
}
```

The output of the program is the following:

```
Class Constructor B1
Class Constructor B2
Class Constructor M1
Class Constructor M2
Class Constructor D
Class Destructor D
Class Destructor M2
Class Destructor M1
Class Destructor B2
Class Destructor B1
```

- If a derived class does not have a defined ***copy-constructor***, this constructor *will be automatically generated by the compiler*.
  - It calls the copy-constructors of the base classes, followed (if necessary) by the copy-constructors (or pseudo-constructors) of the member objects of the class

Example.

```
#include <iostream>
using namespace std;

class Base {
  int n;
public:
  Base(int i): n(i) {
    cout << ″Base(int i)″ << endl;
  }
  Base(const Base& b): n(b.n) {
    cout << ″Base(const Base& b)″ << endl;
  }
  Base(): n(0) { cout << ″Base()″ << endl; }
  void Print() const {
    cout << ″Base; n=‶ << n << endl;
  }
};

class Member {
  int n;
public:
  Member(int i): n(i) {
    cout << ″Member(int i)″ << endl;
  }
  Member(const Member& m): n(m.n) {
    cout << ″Member(const Member& m)″ << endl;
  }
  void Print() const {
    cout << ″Member; n=‟ << n << endl;
  }
};

class Derived: public Base {
  int n;
  Member m;
public:
  Derived(int i): Base(i), n(i), m(i) {
    cout << ″Derived(int i)″ << endl;
  }
  void Print() const {
    cout << ″Derived; n=‟ << n << endl;
    Base::Print();
    m.Print();
  }
};
```

```
int main() {
  Derived o1(7);
  cout << ″ copy-constructor call: ‷ << endl;
  Derived o2 = o1;
  cout << ″Values in o2: ″ << endl;
  o2.Print();
  return 0;
}
```

Program output:

```
   Base(int i)
   Member(int i)
   Derived(int i)
   Copy-constructor call:
   Base(const Base& b)
   Member(const Member& m)
   Values in o2:
   Derived; n=7
   Base; n=7
   Member; n=7
```

- In the case when a *copy-constructor is explicitly defined* in a derived class, *it must call explicitly* (in the constructor initializer list) *the copy-constructor* of the base class.

Example.

A ***wrong*** copy-constructor of the class *Derived*:

```
Derived(const Derived& d): n(d.n), m(d.m) { }
```

The last three lines of program output are:

```
    Derived; n=7
    Base; n=0
    Member; n=7
```

A ***correct*** copy-constructor of the class *Derived*:

```
Derived(const Derived& d): Base(d), n(d.n), m(d.m) { }
```

The program output will be the same as in the last example.

- In ***Python***, constructors are not called automatically (Python is interpreted, not compiled)
- The **`__init__`** function of a *subclass* can call the **`__init__`** functions of its *superclasses*
- In the following example, the **`__init__`** function of class **`B`** calls the **`__init__`** function of its superclass (**`A`**), while **`__init__`** function of the class **`C`** does not call it:

```
class A:
   k = 1
   def __init__(self, a):
      self.a = a

class B(A):
   def __init__(self, a, b):
      super().__init__(a)
      self.b = b

class C(A):
   def __init__(self, c):
      self.c = c
   def f(self):
      print(k)

a = A(7)
b = B(7, 5)
c = C(3)

print('a.dict = ', a.__dict__)
print('b.dict = ', b.__dict__)
print('c.dict = ', c.__dict__)

>>>

a.dict =  {'a': 7}
b.dict =  {'a': 7, 'b': 5}
c.dict =  {'c': 3}
```

- In Python, there is a difference between the memory structure of a *compound object* and the *instance of a subclass*
- Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

class Circle1(Point):
    def __init__(self, x, y, r):
        super().__init__(x, y)
        self.r = r
    def __str__(self):
        return "({0},{1},{2})".format(self.x, self.y,
         self.r)

class Circle2:
    def __init__(self, x, y, r):
        self.p = Point(x, y)
        self.r = r
    def __str__(self):
        return "({0},{1},{2})".format(self.p.x, self.p.y,
         self.r)

c1 = Circle1(3, 5, 1)
c2 = Circle2(2, 4, 1)

print(c1.__dict__)
print(c2.__dict__)
```

```
>>>
{'y': 5, 'r': 1, 'x': 3}
{'r': 1, 'p': <__main__.Point object at 0x0060DB30>}
```

- The **`__init__`** function of a superclass can be called in several ways. For the above class **`B`**:
  - Using the name of the superclass:

    ```
    A.__init__(a)
    ```

  - Using the function **`super()`**:

    ```
    super().__init__(a)       # new-style classes
    super(B, self).__init__(a)
    ```

- In *new-style classes*, the **`super()`** function returns a *proxy object* that allows to call *methods* of a *parent* or a *sibling* class

- **`super()`** uses **`__mro__`** (*method resolution order*):
  - The ***method resolution order*** (*MRO*) is the set of rules that construct the *linearization* of a class
  - The *linearization* of a class ***C*** is the list of the *ancestors* of ***C***, including the class itself, ordered from the nearest ancestor to the furthest

- The syntax:

```
    super().method(args)
```

  is similar to:

```
    super(subclass, instance).method(args)
```

  because for each instance, its associated class can be determined:

```
    super(self.__class__, self).method(args)
```

- For an instance **`a`**, the name of its related class can be determined in two ways:

```
  a.__class__
  type(a)
```

- In new-style classes, both variants are equivalent
  - This type of information (*RTTI*: *Run-Time Type Information*) can be used in C++ only at *runtime*, and in the presence of *polymorphism*

- Other two functions related to *type names* are **`isinstance()`** and **`issubclass()`**:
  - **`isinstance(object, classinfo)`**, which determines if an object is an instance of a class
  - **`issubclass(class, classinfo)`**, which determines if a class is a subclass of another class

```
class A:
    pass

class B(A):
    pass

issubclass(B, A)    # >>> True
isinstance(B, A)    # >>> False

a = A()
b = B()

isinstance(a, A)    # >>> True
isinstance(a, B)    # >>> False
isinstance(a, object) # >>> True
```

C. Public and private inheritance

- The using of public or private inheritance type has a distinct significance in designing and developing an application

C1. Public inheritance

- The ***public inheritance*** is used in the case when a derived class represents conceptually a *specialization* of the base class
- In the literature this relation type is denoted a "***is-a***" relation:
  - The class *student* is a specialization of the class *person*, because every *student* ***is a*** *person*
- This type of inheritance supposes that the derived class *inherits* both the ***interface*** and the ***implementation*** of the base class:
  - In this way an object of the derived class *can be used instead of* an *object* from the *base class*

Example. Considering the classes *Point* and *Circle* from the previous example, the function *Distance* determines the distances between two points:

```
double Distance (Point& p1, Point& p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
         (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}
```

The next sequence displays correct the values 1 and 4:

```
   Point p1(1, 1), p2(0, 0);
   Circle c1(7, 7), c2(3, 3);
   double d1 = Distance(p1, p2);
   double d2 = Distance(c1, c2);
   cout << d1 << d2 << endl;
```

- An important propriety of the inheritance relation is the fact that a class that is public derived from a base class is treated as a ***subtype*** of that base class:
  - All instances of the derived class are compatible with the objects of the base class, and they can be used instead of them.
  - The following sequence is correct:

```
Point p2;
p2 = c1;
```

- In this case, the assignment operation copies *only the members of the base class*
- The previous rule of assignment compatibility is extended also to pointers and object references. For example:

```
class B { /* ... */ };
class A1: public B { /* ... */ };
class A2: public B { /* ... */ };

A1 a1;
A2 a2;
B *pb;
pb = &a1;
pb = &a2;
```

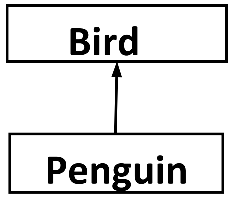
- In the previous example there exists a *default conversion* type from a pointer to a derived class to a pointer to the base class. Such a conversion is denoted ***upcasting***
- The ***upcasting*** operation is frequently used in the applications that use class hierarchies, allowing a uniform treatment of the objects from such a hierarchy by using pointers to the base class
- A public derived class inherits both the *declaration* and the *implementation* of the base class:
  - In the derived class some of the members from the base class can be redefined.
  - The redefinition of a member hides in the derived class the member from the base class with the same name.

Example.

```
class A {
public:
  void f() const {
    cout << ″f in class A″ << endl;
  }
};

class B: public A {
public:
  void f() const {
    cout << ″f in class B″ << endl;
  }
};

int main (){
  A a;
  a.f();        //f from the class A
  B b;
  b.f();        //f from the class B
  b.A::f();     //f from the class A
  // ...
}
```

Remark. The redefinition of the members from a base class in a derived class is not a good idea, because it supposes an error of designing of the class hierarchy. *A better idea* is the using of ***virtual functions*** instead of redefining functions.

Example. There are birds that do not fly, even if the great majority of them flies: for example the penguin.

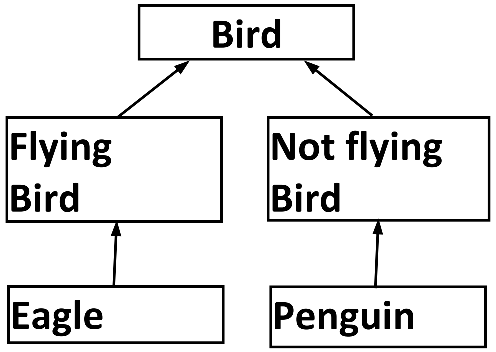


```
class Bird {
public:
  void flies () const {
    cout << ″bird flies″ << endl;
  }
  // ...
};

class Penguin: public Bird { /* ... */ }

Bird eagle;
Penguin penguin;
eagle.flies();        //correct
penguin.flies();      //error!!
```

A way for modify of this code is the redefinition of the *flies* function in the *penguin* class:

```
class Penguin: public Bird {
public:
  void flies () const {
    cout << ”bird does not fly” << endl;
  }
  // ...
};

// ...
Penguin penguin;
penguin.flies();  //correct
```

A *better* solution is a correct design of the class hierarchy that must *distinguish* between the two bird categories:



```
class FlyingBird: public Bird {
public:
  void flies () const {
    cout << ″bird flies″ << endl;
  }
  // ...
};

class NotFlyingBird: public Bird {
public:
  void flies () const {
    cout << ″bird does not fly″ << endl;
  }
  // ...
};

class Eagle: public FlyingBird {
  // ...
};

class Penguin: public NotFlyingBird {
  //...
};
```

```
// ...
Eagle eagle;
Penguin penguin;
Eagle.flies();
```

- In ***Python*** there is only public inheritance, and then the ***subtyping*** relation between a subclass and its superclass can be considered as be present

```
import math
class Point1:
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Circle1(Point1):
    def __init__(self, x, y, r):
        super().__init__(x, y)
        self.r = r
def dist(p1, p2):
    return math.sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2))
p1 = Point1(0, 0)
p2 = Point1(1, 1)
c1 = Circle1(0, 0, 1)
c2 = Circle1(1, 1, 1)
d1 = dist(p1, p2)
d2 = dist(c1, c2)
print(d1, d2)    # d1==d2
```

- In fact, ***Python*** has no ***strong typing***, so the above example is not a really ***subtyping*** relation
- Like other dynamic languages, Python has a so-called ***duck typing*** (or ***automatic interfaces***)
- ***Duck typing*** is a feature of a type system where the ***semantics*** of a class is determined by his ***ability to respond to some message*** (method or property)
- The name comes from the phrase "***If it looks like a duck and quacks like a duck, then it probably is a duck***"
- With normal typing, suitability is determined by an object's type. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself
- ***Duck typing*** is similar to, but distinct from ***structural typing***:
  - ***Structural typing*** is a ***static typing*** system that determines type compatibility and equivalence by a type's structure, whereas ***duck typing*** is ***dynamic*** and determines type compatibility by only that part of a type's structure that is accessed during ***run time***

C2. Private inheritance

- In ***private inheritance***, all the members of the base class become *private* in the derived class (they are not longer available outside of the derived class).
- In this way, a derived class inherits *only the implementation* of the base class, and *not its interface*.
  - This kind of relation is called in the literature as "***an_implementation_of***" relation
- In the case of private inheritance an object from the derived class *is not converted* by the compiler to a base class object (the derived class does not represent a subtype of the base class).

Example. A new perspective of the previous defined classes, *Point* and *Circle*.

```
class Point {
protected:
  double x, y;
public:
  void SetCoord (double a, double b) {
    x = a;
    x = b;
  }
  Point (double a = 0, double b = 0) {
    SetCoord(a, b);
  }
  double X() const { return x; }
  double Y() const { return y; }
};

class Circle: Point {
public:
  double r;
  void SetRadius (double a = 1) { r = a; }
  Circle (double a = 0, double b = 0, double c = 1):
        Point(a,b),r(c) { }
};


double Distance(Point p1, Point p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
          (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}

// ...
point p1(0, 0), p2(1, 1);
circle c1(3, 3), c2(7, 7);
```

```
double d1 = Distance(p1, p2); //correct
double d2 = Distance(c1, c2); //error!!!
```

Remark. The same type of relation, ***is_an_implementation_of***, is also represented by the *objects composition* (for a sub-object of a compound class, only the implementation of the class to which the sub-object belongs to is inherited, and not its interface):

- Usually, it *is indicated to use objects composition*, every time it is possible, instead of using private inheritance
- The using of *private inheritance is necessary only* in the cases when *the base class contains protected members* that otherwise cannot be used in the derived class

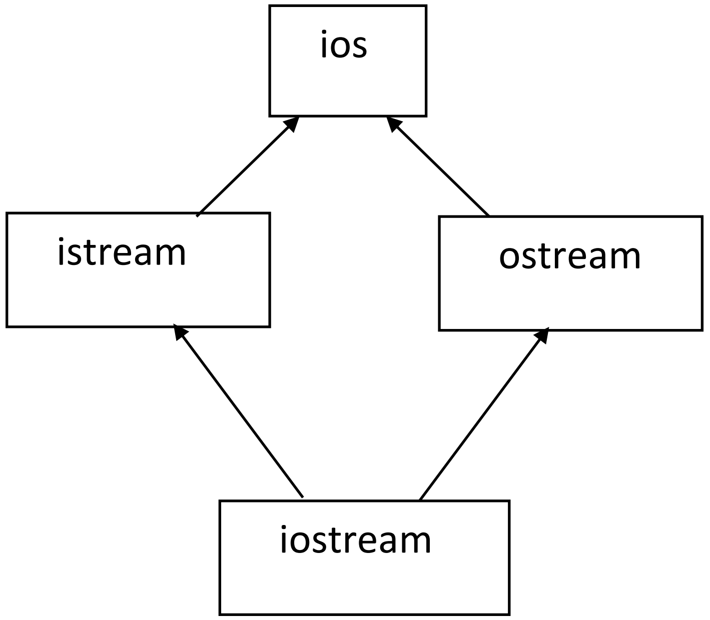
Example. Defining a class, *OrderedList*, which stores the elements of a list in an increasing order, by using the already defined class *list*.

```
class OrderedList: list {
public:
  OrderedList();
  ~ OrderedList();
  void AddOrdered (int);
  void Print();
};
```

- In ***Python*** there is not *private inheritance*

D. Multiple inheritance

- ***Multiple inheritance***: when a derived class inherits from more than one base classes
- Not all the languages which support the object-oriented programming paradigm accept multiple inheritance
- Example: *Smalltalk*, a *pure object-oriented language*, has a predefined class hierarchy representing a tree with only one root called *Object*.
  - Every class defined by the programmer must be derived from *Object* or from a class derived from *Object*
- The ***C++ language*** is *not a pure object-oriented language* (it is a *hybrid* language), but it has the advantage of allowing the creation of classes and classes hierarchies, independent of a certain predefined hierarchy
- An example of ***multiple inheritance*** exists in the predefined classes for Input/Output operations (defined in the *iostream* header file). The diagram of this hierarchy is presented in the following figure:



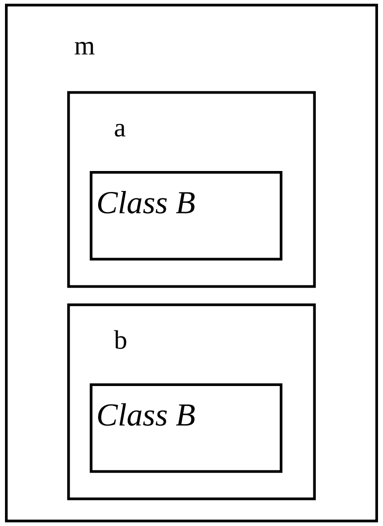
- There are two important problems which can appear in the case of using multiple inheritance:
    - the ***duplication of hidden objects***
    - the ***existence of some members with the same name*** in the different base classes
- The duplication of hidden objects can appear in the case of hierarchies that has a structure as presented in the previous figure
    - the general case: when there *are several paths between a derived class and a base class*, there is *a multiplied hidden object for each path*

Example:

```
class B {
  // ...
};

class M1: public B {
  int a;
  // ...
};
```

```
class M2: public B {
  int b;
  // ...
};

class MI: public M1, public M2
{
  int m;
  // ...
};
```

an object of the class *MI* has the following structure, as presented in the following figure:



- There are *two identical hidden sub-objects*, corresponding to the same class *B*. This fact produces an additional amount of memory, which can induce ambiguities in an application which it is not correct designed
- The second problem of multiple inheritance is generated by the case in which *several base classes have a member with the same name*: how this member will be used in the derived class?

Example:

```
class B1 {
public:
  int a;
  // ...
};
```

```
class B2 {
public:
  double a;
  // ...
};
```

```
class D: public B1, public B2 {
  // ...
};

void Processing () {
  D d;
  d.a = 5;     // error!!
}
```

- In this case there exist *two common variants* of avoiding the confusion:
    - the *explicit use of a certain member* by using the *resolution operator*
    - the *redefinition in the base class* of a *member with the same name*
- *In the first case* the necessity of writing correct code is exclusively the task of the *programmers who uses a class hierarchy* already designed. For example, the statement for the previous example that generates the error could be written as follows:

```
d.B1::a = 5;
```

- *The second case* represents a problem for the *programmers who design the class hierarchy*

Example: A class hierarchy that defines a class *CircleText* thate allows displaying a text in the circle (the implementations of member functions that are not `inline` were not specified):

```
class Point {
protected:
  int x, y;
public:
  Point(int a, int b): x(a), y(b) { }
  int X() const { return x; }
  int Y() const { return y; }
};

// graphic point
class GPoint: public Point {
protected:
  int visible;
public:
  GPoint(int a, int b): Point(a, b), visible(1)
   {   }
  // displays a graphic point on the screen
  void Show();
  void Hide() { visible = 0; }
  int IsVisible() const { return visible; }
  void Translate(int a, int b) { x = a; y = b; }
};

class Circle: public GPoint {
protected:
  int r;
public:
  Circle(int a, int b, int c): GPoint(a, b), r(c) { }
  void Show();    //draw a circle
};

//displays a message on the screen in a rectangle
class Message: public Point {
public:
  char *msg;      //message
  int l, L;       //rectangle dimensions
  Message(int a, int b, int c, int d, char *m):
    Point(a, b), l(c), L(d), msg(m) { }
  void Show();    //displays the message
};
```

```
class CircleText: Circle, Message {
public:
  CircleText(int a, int b, int r, char *m):
    Circle(a, b, r), Message(a, b, r, r, m) { }
  void Show()  //displays the circle with the message inside
  {
    Circle::Show();
    Message::Show();
  }
};

int main() {
  // ...
  CircleTexT c1(250, 100, 25, ″circle C1″);
  c1.Show();
  // ...
}
```

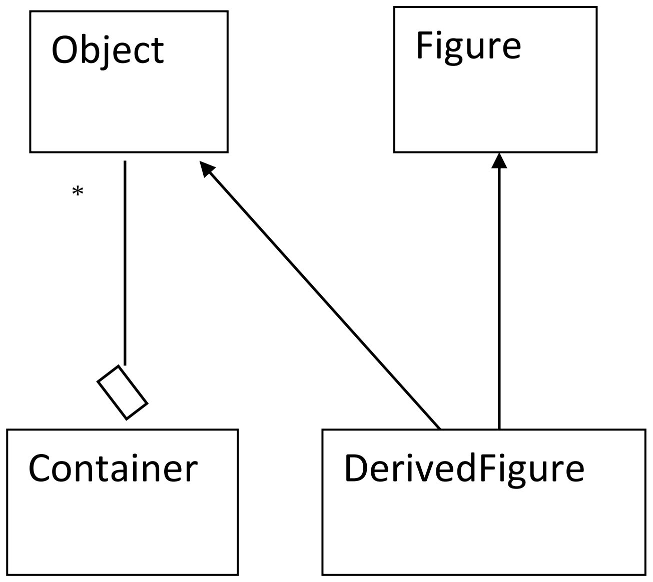
- In the previous class hierarchy the multiple inheritance it is not absolutely necessary. A more real alternative can be the class *CircleText* as a composed class which contains inside two private objects, instances of the *Circle* and *Message* classes.

```
class CircleText {
  Circle circle;
  Message message;
public:
  CircleText(int a, int b, int r, char *m):
    circle(a, b, r), message(a, b, r, r, m) { }
  // displays the message in the circle
  void Show()
  {
    circle.Show();
    message.Show();
  }
}
```

- The necessity of using multiple inheritance can appear when a derived class needs the control over the base classes (by using the **`virtual`** mechanism, for example)

Example

- Another example is referring to the necessity of using some class hierarchies for which the sources cannot be modified (there are available only libraries and header files)
- Consider a class *Container*, which stores the pointers to a list of abstract objects from the *Container* class. In order to be able to use the *Container* class in an application for storing objects belonging to another class, *Figure*, where the application cannot have access, a new class, *DerivedFigure*, can be created by inheriting both the *Object* and *Figure* classes, as presented in the next figure.



In this way, by using the *upcasting* mechanism, the container can store also references to the objects of the class *DerivedFigure*

- Like *C++* but unlike *Java*, ***Python*** allows ***multiple inheritance***:

```
class A: pass

class B: pass

class C(A, B): pass
```

- Multiple inheritance can pose a *conflict* when the same method (or other attribute) name is defined in more than one superclass

- Such a conflict is resolved either *automatically* by the inheritance search order, or *manually* in the code:
  - By *default*, inheritance chooses the *first occurrence* of an attribute it finds when an attribute is referenced normally

    ```
    self.method(), self.attr
    ```
  - In this mode, Python chooses the lowest and leftmost in classic classes, and in non-diamond patterns in all classes
    - new-style classes may choose an option to the right in diamond patterns
  - *Explicit*: In some class models, an attribute can be selected explicitly by referencing it through its class name:

    ```
    superclass.method(self), superclass.attr
    ```

- The *search order* for an attribute of a class (the ancestor tree) is determined by the *Method Resolution Order* (*MRO*) algorithm
- This ordered list can be retrieved by using the **`__mro__`** attribute:

```
class A: pass
class B: pass
class C(A, B): pass
class D: pass
class E(C, D): pass

print(E.__mro__)

(<class '__main__.E'>, <class '__main__.C'>,
 <class '__main__.A'>, <class '__main__.B'>,
 <class '__main__.D'>, <class 'object'>)
```

- The ***C3 Method Resolution Order***:
  - *The linearization of* ***C*** *is the sum of* ***C*** *plus the merge of the linearizations of the parents and the list of the parents*:

    $L[C(B_1 \ldots B_N)] = C + merge(L[B_1] \ldots L[B_N], B_1 \ldots B_N)$

    L[object] = object

- For example:

```
O = object

class F(O): pass
class E(O): pass
class D(O): pass
class C(D,F): pass
class B(D,E): pass
class A(B,C): pass

L[O] = O
L[D] = D O
L[E] = E O
L[F] = F O
```

```
L[B] = B + merge(DO, EO, DE)
     = B + D + merge(O,EO,E)
     = B + D + E + merge(O,O)
     = B D E O
L[C] = C + merge(DO,FO,DF)
     = C + D + merge(O,FO,F)
     = C + D + F + merge(O,O)
     = C D F O
L[A] = A + merge(BDEO,CDFO,BC)
     = A + B + merge(DEO,CDFO,C)
     = A + B + C + merge(DEO,DFO)
     = A + B + C + D + merge(EO,FO)
     = A + B + C + D + E + merge(O,FO)
     = A + B + C + D + E + F + merge(O,O)
     = A B C D E F O
```

- The “*Diamond problem*”:

```
O = object
class A(O): pass
class B(O): pass
class C(A,B): pass

L[A] = A O
L[B] = B O
L[C] = C + merge(AO, BO, AB)
     = C + A + merge(O, BO, B)
     = C + A + B + merge(O,O)
     = C A B O
```

- With multiple inheritance, ***`super()`*** should not see as a *function call* to the next “up” in the *inheritance chain*
  - When properly used, **`super()`** will ensure that *all functions* in the MRO are called in that order

```
class A(object):
  def __init__(self):
    super().__init__()
    print('A')

class B(A):
  def __init__(self):
    super().__init__()
    print('B')
```

```
class C(object):
  def __init__(self):
    super().__init__()
    print('C')

class D(B, C):
  def __init__(self):
    super().__init__()
    print('D')

d = D()  # C A B D
```