EPFL Fall 2022
Master level
MATH-458
September 22, 2023

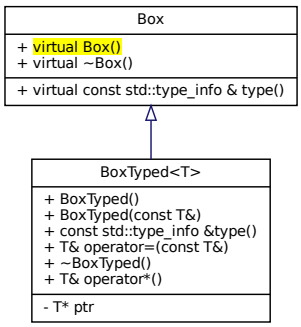
Programming Concepts in Scientific Computing

"Box" and "Any" classes

The goal of the present exercises is to employ templates to make generic objects capable of storing any object type.

Exercise 1: *Box*

A Box family of objects is aimed at storing typed pointers together with the information of the type. We will employ the following class family, with one mother class and its templated daughter class.



- The 'type' method allows to get a type_info object (please seek information on c++ reference website).
- The operator "*" allows one to fetch the stored object
- If implemented correctly we should be able to do:

```
BoxTyped<int> box;
*box = 2;
Box &anonymous_box = box;
std::cout << anonymous_box.type().name() << std::endl;
```

  as well as:

```
A a; // some object a with type A being a class
BoxTyped<A> box(a);
Box &anonymous_box = box;
std::cout << anonymous_box.type().name() << std::endl;
```

1. Implement the family of classes.
2. Beware memory leaks.

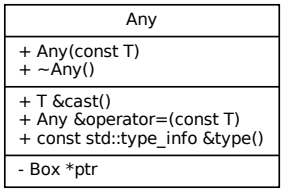
3. Implement a variadic template version for the constructor of BoxTyped which will forward the arguments to the constructor of the "T" type. It should then become possible to do:

```
class B{
public:
 B(int a, double b){};
};

BoxTyped<B> box2(1, 2.);
Box &anonymous_box = box;
std::cout << anonymous_box.type().name() << std::endl;
```

Exercise 2: *Any*

Let us consider the non templated class "Any" (beware, the template parameters are for the methods and not for the class).



Since "Any" is not templated it can store seamlessly any type and can be used in the following way:

```
Any i = 2;
Any a = A{};
Any b = B{2};
```

1. Implement what is needed to fetch the type info of the stored object:

```
std::cerr << "i type: " << i.type().name() << std::endl;
std::cerr << "a type: " << a.type().name() << std::endl;
std::cerr << "b type: " << b.type().name() << std::endl;
```

2. The "Any" class should allow to retrieve the real object with the templated method "cast". By using the templated function "dynamic_cast", please implement what is needed to allow writing the code below. (have a look to C++ reference for dynamic_cast).

```
std::cerr << "i cast: " << i.cast<int>() << std::endl;
std::cerr << "a cast: " << &a.cast<A>() << std::endl;
std::cerr << "b cast: " << &b.cast<int>() << std::endl;
```

3. What happens if you try to cast to a wrong type ?