- Examples

In this lesson, we'll look at a couple of examples of type erasure.

WE'LL COVER THE FOLLOWING

Example 1: Type Erasure Using Object-Oriented Programming

```cpp
// typeErasureOO.cpp

#include <iostream>
#include <string>
#include <vector>

struct BaseClass{
        virtual std::string getName() const = 0;
};

struct Bar: BaseClass{
        std::string getName() const override {
                return "Bar";
        }
};

struct Foo: BaseClass{
        std::string getName() const override{
                return "Foo";
        }
};

void printName(std::vector<BaseClass*> vec){
    for (auto v: vec) std::cout << v->getName() << std::endl;
}
```

```
int main(){

        std::cout << std::endl;

        Foo foo;
        Bar bar;

        std::vector<BaseClass*> vec{&foo, &bar};

        printName(vec);

        std::cout << std::endl;
}
```

Explanation

The key point is that you can use instances of `Foo` or `Bar` instead of an instance for `BaseClass`. `std::vector<BaseClass*>` (line 35) has a pointer to `BaseClass`. Well, actually it has two pointers to `BaseClass` (derived) objects and it is a vector of such pointers. `BaseClass` is an abstract base class, which is used in line 23. `Foo` and `Bar` (lines 11 and 17) are the concrete classes.

What are the pros and cons of this implementation with object-orientation?

Pros:

- Type-safe
- Easy to implement

Cons:

- Virtual dispatch
- Intrusive because the derived class must know about its base class

Example 2: Type Erasure with Templates

```
// TypeErasure.cpp

#include <iostream>
#include <memory>
#include <string>
#include <vector>

class Object {

public:
    template <typename T>
```

```
    explicit Object(const T& obj): object(std::make_shared<Model<T>>(std::move(obj))){}

    std::string getName() const {
        return object->getName();
    }

   struct Concept {
       virtual ~Concept() {}
           virtual std::string getName() const = 0;
   };

   template< typename T >
   struct Model : Concept {
       explicit Model(const T& t) : object(t) {}
           std::string getName() const override {
                   return object.getName();
           }
     private:
       T object;
   };

   std::shared_ptr<const Concept> object;
};


void printName(std::vector<Object> vec){
    for (auto v: vec) std::cout << v.getName() << std::endl;
}

struct Bar{
        std::string getName() const {
            return "Bar";
        }
};

struct Foo{
        std::string getName() const {
            return "Foo";
        }
};

int main(){

        std::cout << std::endl;

        std::vector<Object> vec{Object(Foo()), Object(Bar())};

        printName(vec);

        std::cout << std::endl;
}
```

Explanation

First of all, the `std::vector` uses instances (line 57) of type `Object` and not pointers like in the previous object-oriented example. These instances can be created with arbitrary types because it has a generic constructor (line 12). The

`Object` has the `getName` method (line 14) which is directly forwarded to the `getName` of the object. `object` is of type `std::shared_ptr<const Concept>`. The emphasis should not lay on the empty but on the `virtual` destructor in line 19. When the `std::shared_ptr<const Concept>` goes out of scope, the destructor is called. The `static` type of object is `std::shared_ptr<const Concept>` but the dynamic type `std::shared_ptr<Model<T>>`. The `virtual` destructor guarantees that the correct destructor is called. This means, in particular, the destructor of the dynamic type. The `getName` method of `Concept` is pure virtual (line 18), therefore, due to virtual dispatch, the `getName` method of `Model` (line 24) is used. In the end, the `getName` methods of `Bar` and `Foo` (lines 42 and 48) are applied in the `printName` function (line 37).

In the next lesson, we'll solve an exercise on type erasure.