Idioms and Patterns: Type Erasure

In this lesson, we'll learn about type erasure in detail.

WE'LL COVER THE FOLLOWING

Type Erasure

Type Erasure enables you to use various concrete types through a single generic interface.

> Type erasure is duck typing applied in C++

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." (James Whitcomb Rileys)

Of course, you've already used type erasure in C++ or C. The C-ish way of type erasure is a `void` pointer; the C++ish way of type erasure is object-orientation.

Typical Use Case

Type erasure can be performed with `void` pointers, object-orientation, or templates.

Case 1: Void Pointers

```
void qsort(void *ptr, std::size_t count, std::size_t size, cmp);
```

Before the `qsort()`, consider using:

```
int cmp(const void *a, const void *b);
```

The comparison function `cmp` should return a

- **negative integer:** the first argument is less than the second
- **zero:** both arguments are equal
- **positive integer:** the first argument is greater than the second

Thanks to the `void` pointer, `std::qsort` is generally applicable but also quite error-prone.

Maybe you want to sort a `std::vector<int>`, but you used a comparator for C-strings. The compiler cannot catch this error because the type of information was removed and we end up with undefined behavior.

In C++ we can do better.

Case 2: Object-Orientation

Having a class hierarchy and using the `BaseClass` pointer instead of concrete types is one way to enable type erasure

```
std::vector<const BaseClass*> vec;
```

The `vec` object has a pointer to a `constant BaseClasses`.

Case 3: `std::function`

`std::function` as a polymorphic function wrapper is a nice example of type erasure in `C++`. `std::function` can accept everything, which behaves like a function. To be more precise, this can be any callable such as a function, a function object, a function object created by `std::bind`, or just a lambda function.

Let's have a look at the difference between the three types listed above in the following table:

Technique	Each	Type-safe	Easy to	Common

Technique	Datatype	Interface	implment	Base Class
void*	Yes	No	Yes	No
Object-Orientation	No	Yes	Yes	Yes
Templates	Yes	Yes	No	No

```
class Object {
  public:
  ...
  struct Concept {
    virtual ~Concept() {}
  };

  template< typename T >
  struct Model : Concept {
    ...
  };

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

The small code snippet shows the structure of type erasure with templates.

The components are:

- `Object`: Wrapper for a concrete type
- `object`: Pointer to the concept
- `Concept`: Generic interface
- `Model`: Concrete class

Don't be irritated by the names `Object`, `object`, `Concept`, and `Model`. They are typically used for type erasure in the literature, so we stick to them. We will see a more explained version of this in the example in the next lesson.

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