Polymorphism

In this lesson we deal with the concept of using std::vector and std::variant in Polymorphism. Interesting? Read on to find out more!

WE'LL COVER THE FOLLOWING

Most of the time in C++, we can safely use runtime polymorphism based on a `vtable` approach. You have a collection of related types that share the same interface, and you have a well defined virtual method that can be invoked.

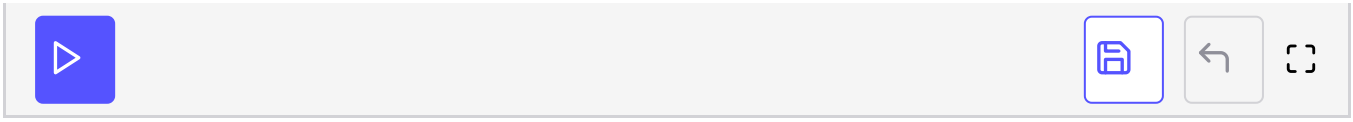
But what if you have “unrelated” types that don’t share the same base class? What if you’d like to quickly add new functionality without changing the code of the supported types?

Example:

With `std::variant` and `std::visit` we can build the following example:

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

class Triangle
{
public:
    void Render() {
        std::cout << "Drawing a triangle!\n";
    }
};

class Polygon
{
public:
    void Render() {
        std::cout << "Drawing a polygon!\n";
    }
};
```

```
class Sphere
{
public:

    void Render() {
        std::cout << "Drawing a sphere!\n";
    }
};

int main()
{
    std::vector<std::variant<Triangle, Polygon, Sphere>> objects {
        Polygon(),
        Triangle(),
        Sphere(),
        Triangle()
    };
    auto CallRender = [](auto& obj) {
        obj.Render();
    };
    for (auto& obj : objects)
        std::visit(CallRender, obj);
}
```



The above example shows only the first case of invoking a method from unrelated types. It wraps all the possible shape types into a single variant and then uses a visitor to dispatch the call to the proper type.

If you'd like, for example, to sort objects, then you can write another visitor, one that holds some state. And that way you'll get more functionality without changing the types.

Now that you've reached the end of the chapter, the next section will give you a summary of all the important concepts you've learned in this chapter along with compiler support for std::variant.