

# Introduction

Let's take a look at what you'll learn in this chapter!

## WE'LL COVER THE FOLLOWING



- Why Use `std::variant`
- What you'll learn in this chapter

## Why Use `std::variant` #

Another handy wrapper type that we get in C++17 is `std::variant`. This is a type-safe union - you can store different type variants with the proper object lifetime guarantee. The new type offers a huge advantage over the C-style union. You can store all of the types inside - no matter if it's something simple like `int`, or `float`, but also complex entities like `std::vector<std::string>`. In all of the cases, objects will be correctly initialised and cleaned up.

What's crucial is the fact that the new type enhances the implementation of design patterns. For example, you can now use a visitor, pattern matching and runtime polymorphism for unrelated type hierarchies in a much easier way.

## What you'll learn in this chapter #

- What problems can occur with unions.
- What discriminated unions are, and why we need type-safety with unions.
- How `std::variant` works and what it does.
- Operations on `std::variant`.

- Performance cost and memory requirements.
- Example use cases

---

Head over to the next lesson to get started with the basics.