# Introduction

The [[nodiscard]] attribute is a way of conveying our intentions to the compiler. We'll find out more in this section.

C++17 brought a few more standard attributes. By using those extra annotations, you can make your code not only readable for other developers, but also the compiler can use this knowledge.

For example, it might produce more warnings about potential mistakes.

Or the opposite: it might avoid a warning generation because it will notice a proper intention (for example with `[[maybe_unused]]`).

In this chapter, you'll see how one attribute - `[[nodiscard]]` - can be used to provide better safety in the code.

The `[[nodiscard]]` attribute was mentioned in the Attributes chapter, but here's a simple example to recall its properties.

The attribute is used to mark the return value of functions:

```
[[nodiscard]] int Compute();
```

When you call such function and don't assign the result:

```
void Foo() {
  Compute();
}
```

You should get the following (or a similar) warning:

```
warning: ignoring return value of 'int Compute()', declared with attribut
e nodiscard
```

We can go further and not just mark the return value, but a whole type:

```
[[nodiscard]] struct SuperImportantType { }
SuperImportantType CalcSuperImportant();


SuperImportantType OtherFoo();
SuperImportantType Calculate();
```

and you'll get a warning whenever you call any function that returns `SuperImportantType`.

In other words, you can enforce the code contract for a function, so that the caller won't skip the returned value. Sometimes such omission might cause you a bug, so using `[[nodiscard]]` will improve code safety.

> The compiler will generate a warning, but usually it's a good practice to enable "treat warnings as errors" when building the code. `/WX` in MSVC or `-Werror` in GCC. Errors stop the compilation process, so a programmer needs to take some action and fix the code.

---

Next, we'll examine some of the use cases where `[[nodiscard]]` is useful.