Where Can It Be Used?

This lesson showcases some situations where the use of [[nodiscard]] is beneficial.

WE'LL COVER THE FOLLOWING

Attributes are a standardised way of annotating the code. They are optional, but they might help the compiler to optimize code, detect possible errors or just clearly express the programmer's intentions.

Here are a few places where `[[nodiscard]]` can be potentially handy:

Errors

One crucial use case for `[[nodiscard]]` are error codes.

How many times have you forgotten to check a returned error code from a function? (Crucial if you don't rely on exceptions).

Here's some code:

```
enum class [[nodiscard]] ErrorCode {
  OK,
  Fatal,
  System,
  FileIssue
};
```

And if we have several functions:

```
ErrorCode OpenFile(std::string_view fileName);
```

```
ErrorCode SendEmail(std::string_view sendto, std::string_view text);
ErrorCode SystemCall(std::string_view text);
```

Now every time you'd like to call such functions you're "forced" to check the return value.

Often you might see code where a developer checks only a few function calls, while other function invocations are left unchecked. That creates inconsistencies and can lead to some severe runtime errors.

You think your method is doing fine (because N (of M) called functions returned `OK`), but something is still failing. You verify it with the debugger, and you notice that the Y function returns `FAIL`, and you haven't checked it.

Should you use `[[nodiscard]]` to mark the error type or maybe some essential functions only?

For error codes that are visible through the whole application that might be the right thing to do. Of course when your function returns just `bool` then you can only mark the function, and not the type (or you can create a `typedef`/`alias` and then mark it with `[[nodiscard]]`).

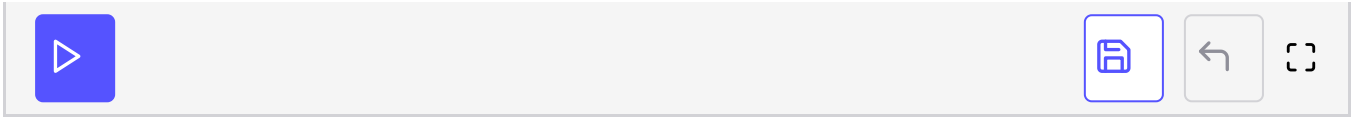
Factories / Handles

Another important type of functions where `[[nodiscard]]` adds value are "factories".

Every time you call "make/create/build" you don't want to skip the returned value. Maybe that's a very obvious thing, but there's a possibility (especially when doing some refactoring), to forget, or comment out.

```
#include <string>

class Foo {
public:
    Foo(std::string param1, std::string param2) { }

    void Run() { }
};

[[nodiscard]] Foo BuildFoo() {
    std::string param1;
    std::string param2;
    return Foo(param1, param2);
}

int main() {
```

```
  // BuildFoo();  // error: return value is being ignored here
    auto foo = BuildFoo();
    foo.Run();


    return 0;
}
```



When Returning Non-Trivial Types?

What about such code:

```
std::vector<std::string> GenerateNames();
```

The returned type seems to be heavy, so usually, it means that you have to use it later. On the other hand, even `int` might be heavy regarding the semantics of the given problem.

Code With No Side Effects?

The code in the previous section:

```
std::vector<std::string> GenerateNames(); // no side effects...
```

This is also an example of a function with no side effects - no global state is changed during the call. In that case, we need to do something with the returned value. Otherwise, the function call can be removed/ optimized from the code.

From the look of things, it may seem that `[[nodiscard]]` can be used anywhere. In the next lesson, we'll understand why this is not the case.