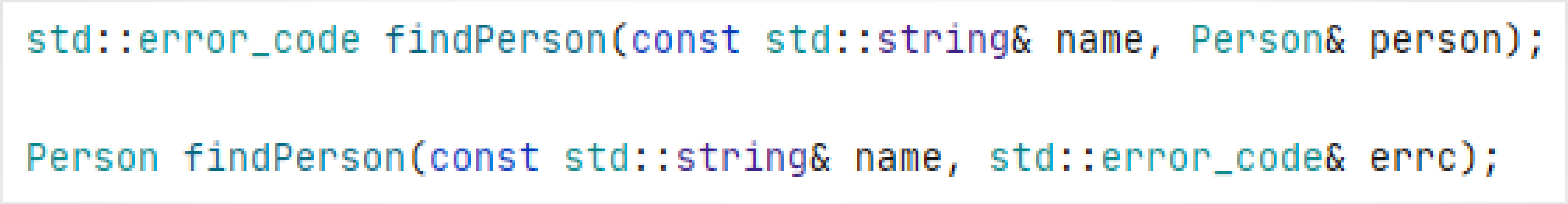
Expected-like types

Error codes aren’t the only alternative to exceptions

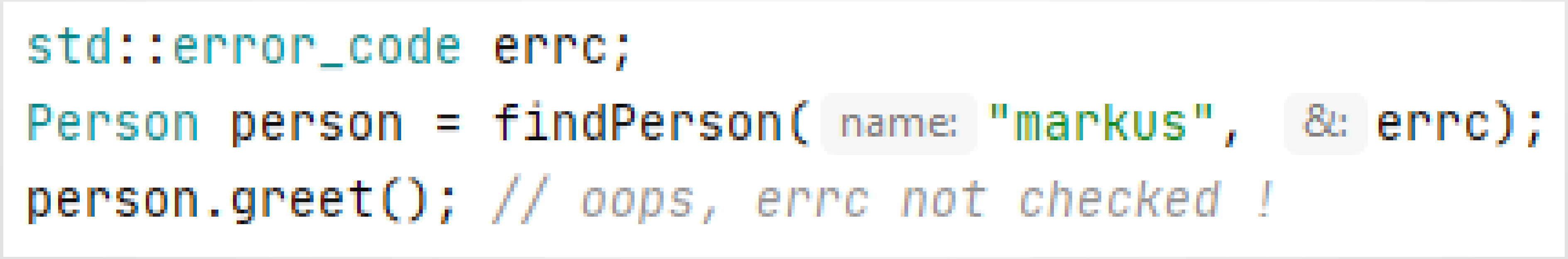
Index

Error codes



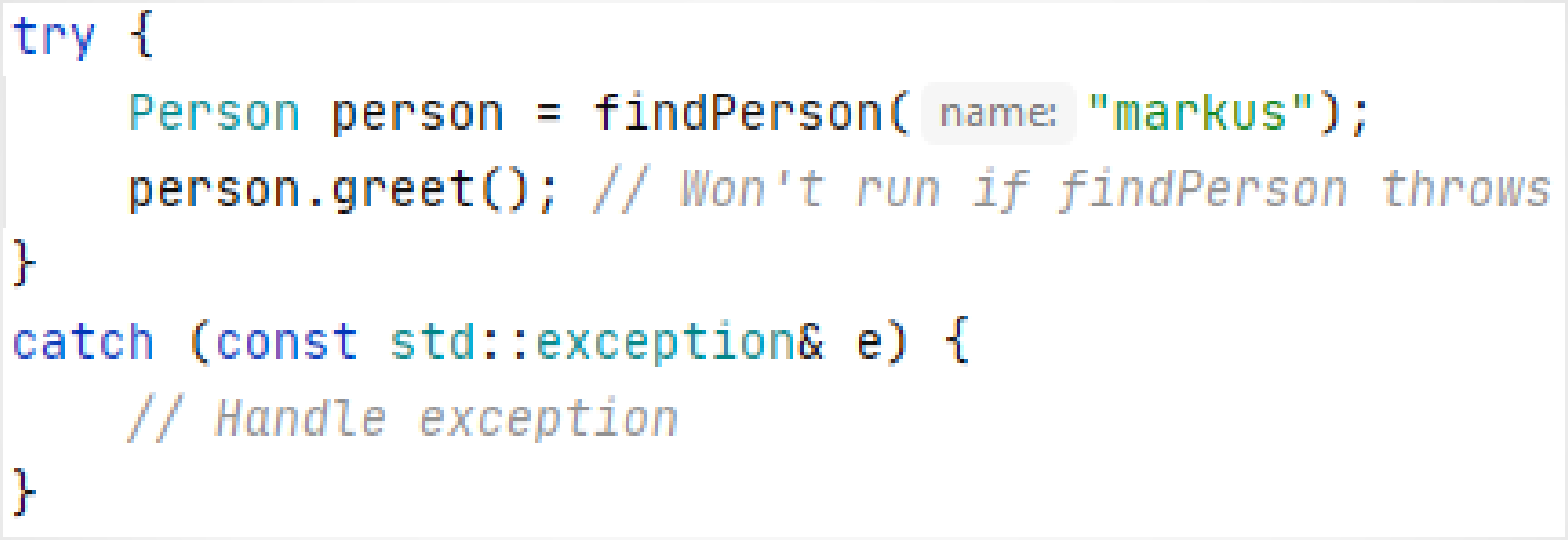
```cpp
std::error_code findPerson(const std::string& name, Person& person);

Person findPerson(const std::string& name, std::error_code& errc);
```

- It’s annoying to use output parameter
- Person must be default-constructible
- It’s easy to not check the error code returned



```cpp
std::error_code errc;
Person person = findPerson( name: "markus",  &: errc);
person.greet(); // oops, errc not checked !
```

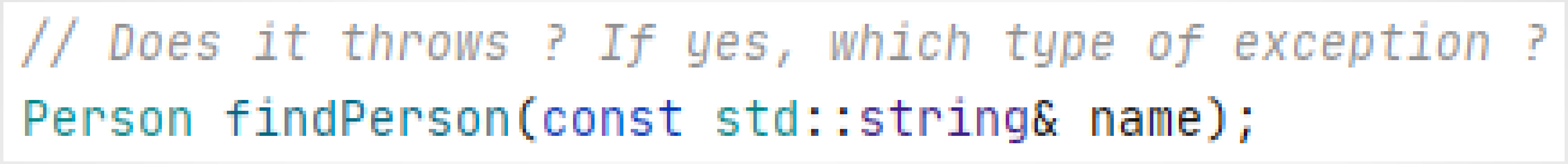
Exceptions



```
try {
    Person person = findPerson( name: "markus");
    person.greet(); // Won't run if findPerson throws
}
catch (const std::exception& e) {
    // Handle exception
}
```

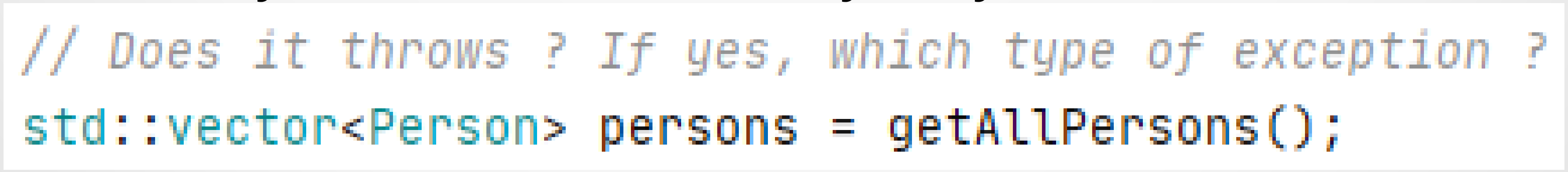
- Functions are easier to use because no output parameters
- Safer than error code : if an exception is thrown the rest of your code won’t run unless you catch the exception

Exceptions



```cpp
// Does it throws ? If yes, which type of exception ?
Person findPerson(const std::string& name);
```

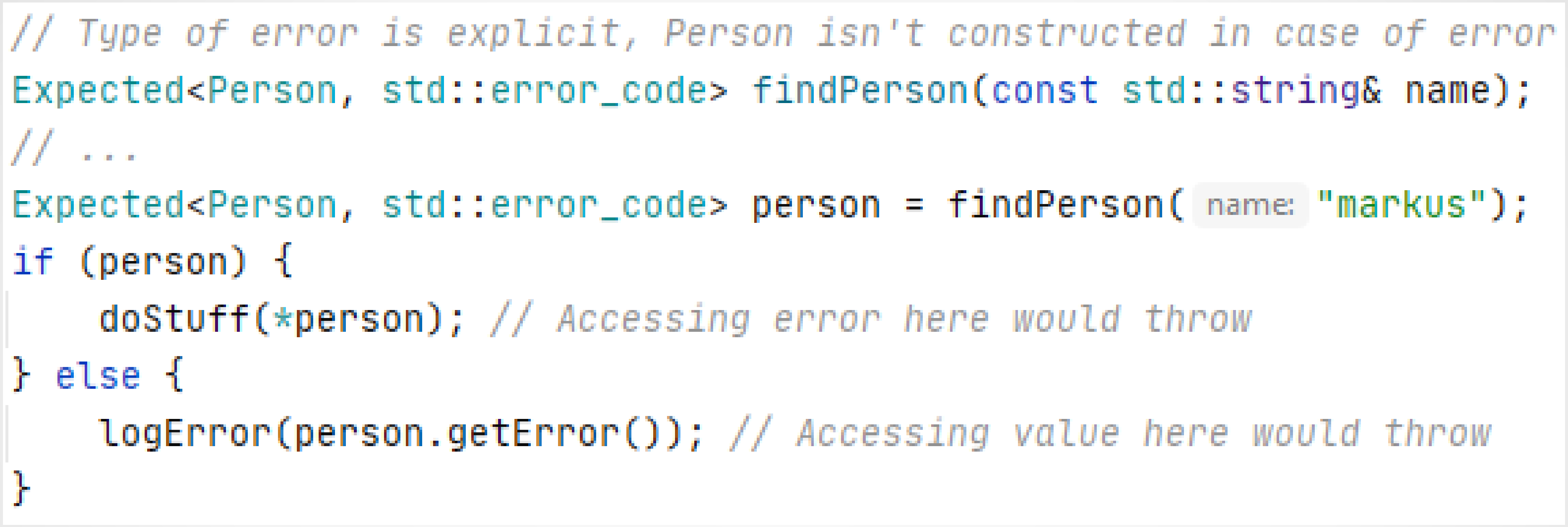
- It’s hard to know if a function throws, you must read the documentation to know it, and it must be up to date
- noexcept(true / false) is rarely used,and it’s annoying to use because you must check the prototype of every function you call to know if they may throw



```cpp
// Does it throws ? If yes, which type of exception ?
std::vector<Person> persons = getAllPersons();
```

- If you want to pass custom information (other than what() string) you need a custom exception type, and the only way to know it can be thrown if through documentation

Expected-like types

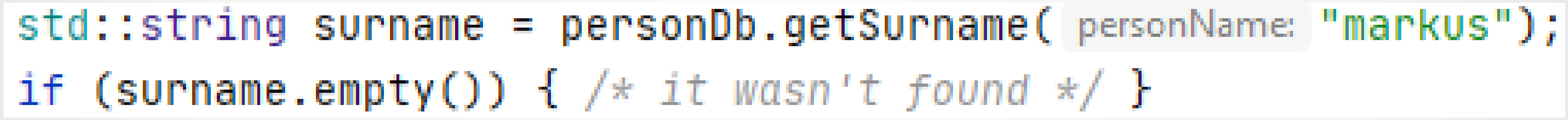


```cpp
// Type of error is explicit, Person isn't constructed in case of error
Expected<Person, std::error_code> findPerson(const std::string& name);
// ...
Expected<Person, std::error_code> person = findPerson( name: "markus");
if (person) {
    doStuff(*person); // Accessing error here would throw
} else {
    logError(person.getError()); // Accessing value here would throw
}
```

- Basically it’s an union / variant of ValueType and ErrorType
- It provide safe access to the value (or error) stored, it throws if it isn’t accessible
- Can be extended a lot to provide a rich and safe API, an example will be showed at the end

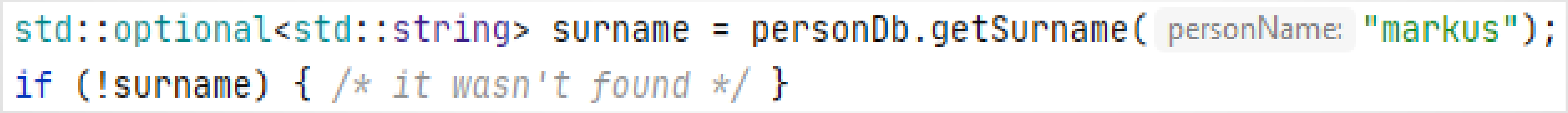
Side note about optional

- If your “error code type” is a boolean, or if you use a sentinel value to check for error (e.g., -1 = error, positive value = real result of function), it’s better to use std::optional



```
std::string surname = personDb.getSurname( personName: "markus");
if (surname.empty()) { /* it wasn't found */ }
```

- This may seem acceptable, but it has the sames issues as error codes, if you don’t check that the value isn’t the sentinel value then you may process invalid data

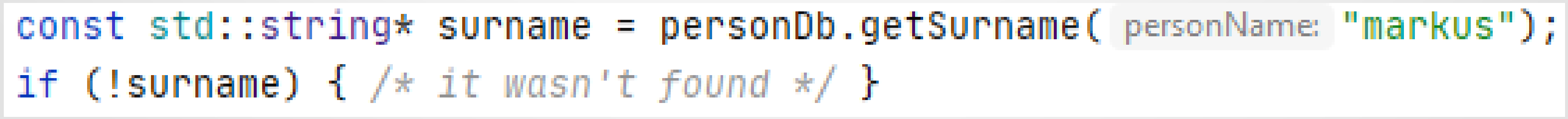


```
std::optional<std::string> surname = personDb.getSurname( personName: "markus");
if (!surname) { /* it wasn't found */ }
```

- The safer solution, if the optional is std::nullopt then you can’t access and missuses the string

Side note about optional

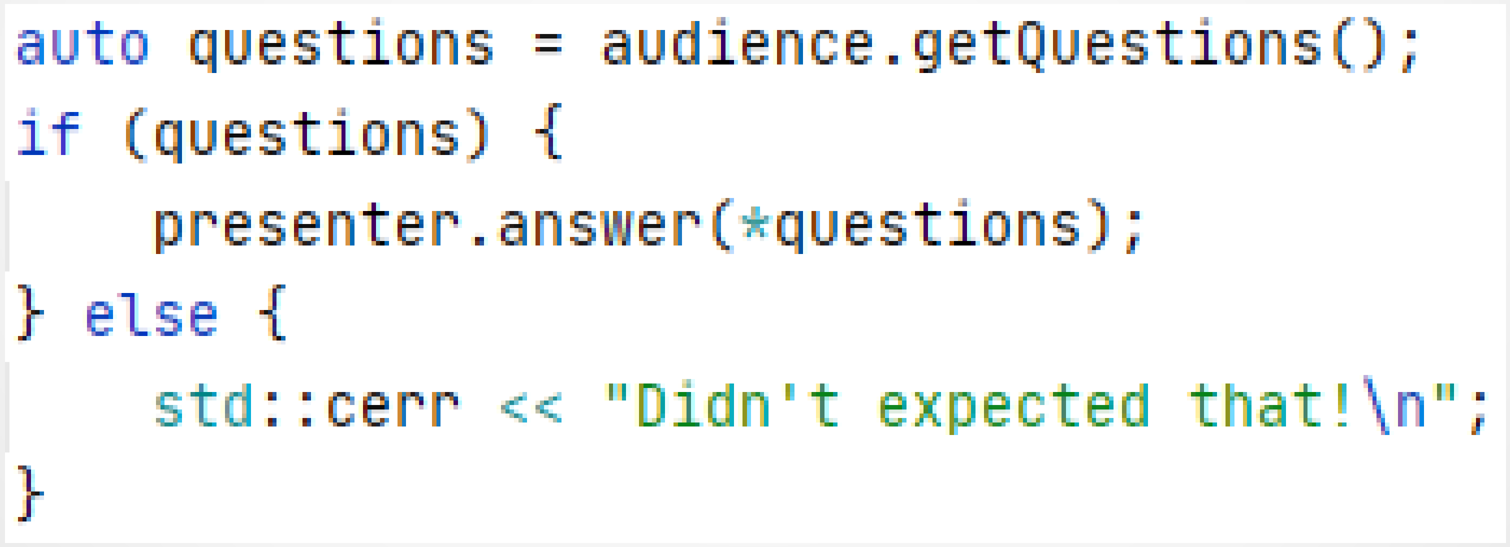
- std::optional as the advantage to be standardized in C++17
- Even if it’s not as feature-rich or as clean as expected-like types, it’s way more safe to use than error codes
- Its main con is that it lacks the ability to give information about the nature of the error, so it’s not suitable for every uses
- You may also use pointers on older C++ versions or for having a pointer-like semantics, but be aware that it brings another kind of issues (use after free, mainly)



```
const std::string* surname = personDb.getSurname( personName: "markus");
if (!surname) { /* it wasn't found */ }
```

Live coding expected-like type

- Any questions before starting ?



```cpp
auto questions = audience.getQuestions();
if (questions) {
    presenter.answer(*questions);
} else {
    std::cerr << "Didn't expected that!\n";
}
```

Live coding expected-like type

Go !

Live coding expected-like type

- Possible safety features:
  - Ensure that the expected was checked before accessing the value / error
  - Ensure that the expected was checked before being destroyed

Live coding expected-like type

- Possible usability features:
  - getValueOrDefault: return the held value or a default value, no need to check the expected before
  - Basic functional functions, like map / then / etc that call functors if some conditions are met