Statyczne wyjątki C++

Każdy język programowania ma swój określony, rekomendowany sposób na obsługę błędów. W Javie są to wyjątki, w języku Rust pomocnicze typy, takie jak Error czy Option. W C z kolei najpopularniejszym sposobem obsługi błędów jest tzw. „kod błędu". A co z językiem C++? Na to pytanie odpowiemy sobie właśnie w tym artykule, zwracając szczególną uwagę na obecne prace grupy standaryzacyjnej w zakresie obsługi błędów.

OBECNY STAN OBSŁUGI BŁĘDÓW W C++

Najlepszym źródłem przykładów tego, w jaki sposób powinniśmy informować użytkowników o błędzie w funkcji, powinna być biblioteka standardowa C++. Pierwsza rzecz, która rzuca się w oczy, to fakt, że jeżeli wyłączymy obsługę wyjątków, to „połowa" funkcji z biblioteki standardowej staje się bezużyteczna – nie mamy możliwości obsługi błędów w ogóle. Dla przykładu, wszystkie kontenery w bibliotece standardowej, które alokują pamięć dynamicznie, używają wyjątków do informowania użytkownika o braku pamięci. Jednocześnie warto zauważyć, że projekty będące OOM-safe (ang. *Out Of Memory Safe* – bezpieczne na brak pamięci) należą do rzadkości (Herb Sutter prowadził eksperyment z rzucaniem `std::bad_alloc`, który jest opisany w dokumencie p0709r2, patrz „Bibliografia" na końcu artykułu).

Można stwierdzić, że pisząc własne funkcje, powinniśmy również używać wyjątków. Nie jest to jednak do końca prawda. Od C++ w wersji 11 biblioteka standardowa zawiera coraz więcej możliwości obsługi błędów – za pomocą kodów błędów. W darmowej wersji C++11 dodano:

- `std::error_code`
- `std::error_category`
- `std::error_condition`

które służą do wygodnego posługiwania się kodami błędów.

W C++17 dodano całą bibliotekę filesystem, która ma „dualne" API, gdzie jedno API wykorzystuje wyjątki do informowania o błędach, a drugie kody błędów.

Co ciekawe, wersja API z kodami błędów również zawiera funkcje, które mogą zwrócić część błędów za pomocą kodu błędu, a część za pomocą wyjątków. Taką funkcją jest na przykład:

```
path absolute(const std::filesystem::path& p,
                   std::error_code& ec);
```

Możemy przeczytać w standardzie, że:

> *$29.11.6.2.2 – file systems -error reporting: Failure to allocate storage is reported by throwing an exception as described in [res.on.exception.handling].*

Jeżeli funkcja `absolute` z jakiegoś powodu nie będzie w stanie zaalokować pamięci, to zamiast zwrócenia odpowiedniej wartości kodu błędu (który de *facto* i tak przyjmuje jako wyjściowy parametr), rzuci wyjątek.

To, że nawet biblioteka standardowa nie jest do końca spójna sama ze sobą, jeżeli chodzi o raportowanie błędów, może wydawać się bardzo zaskakujące.

Przyjrzyjmy się zatem, skąd wywodzą się te niespójności.

OBSŁUGA BŁĘDÓW ZA POMOCĄ WYJĄTKÓW

Być może zaskakującym jest fakt, że około połowa projektów w C++ przynajmniej w części kodu źródłowego ma wyłączoną obsługę wyjątków (dane według: cpp foundation developer survey – patrz „Bibliografia"). Skąd to wynika? Jak nietrudno się domyślić, główną przyczyną rzadkiego używania wyjątków jest ich wydajność. A jeżeli ktoś używa języka C++, to bardzo prawdopodobnym jest, że właśnie na wydajności jego aplikacji mu zależy.

Wydajność wyjątków w C++

Przede wszystkim należałoby powiedzieć, co mamy na myśli, mówiąc „wydajność", bo jak się okazuje, sam fakt, że rzucanie wyjątków jest wolne, nie stanowi aż tak dużego problemu.

Mechanizm rzucania wyjątków jest powolny chociażby ze względu na to, że potrzebuje dopasować odpowiednią klauzulę `catch` do typu rzuconego wyjątku. Co więcej, kompilator musi wygenerować dodatkowe dane (zazwyczaj na stercie), gdzie będzie przechowywać informacje o wyjątku, a także musi zwinąć stos.

Wśród implementatorów kompilatorów, jak i programistów istnieje świadomość tego, że wyjątki sygnalizują wyjątkowy stan aplikacji, w którym nie powinna ona znajdować się zbyt często. W związku z tym kompilatory generują taki kod obsługi wyjątków, aby w przypadku kiedy żaden wyjątek nie jest rzucany, wykonanie kodu było tak szybkie, jak gdyby obsługa wyjątków w ogóle nie istniała. I faktycznie tak jest, o czym można poczytać w raporcie „Technical Report on C++ performance".

Skoro kompilatory są w stanie dobrze optymalizować kod dla przypadków, kiedy wyjątki nie są rzucane, to nasuwa się pytanie: czy wydajność wyjątków naprawdę jest tak ważna, żeby wyłączać wsparcie wyjątków w tylu projektach? W końcu przy dobrze napisanym kodzie zazwyczaj nie będziemy korzystać z wyjątków. Dlatego właśnie najczęściej wyjątki w języku C++ używane są do pisania aplikacji desktopowych, gdzie wydajność nawet jeżeli spadnie, to nie będzie to oznaczać całkowitego spadku responsywności aplikacji.

W większości projektów, gdzie wyjątki nie są dozwolone, nie chodzi tylko o szybkość aplikacji oraz pamięć samą w sobie, ale także o przewidywalność zużycia pamięci oraz przewidywalność czasu obsługi wyjątku. Dotyczy to zwłaszcza systemów czasu rzeczywistego. Okazuje się, że nie jesteśmy w stanie przewidzieć, jak długo zajmie nam obsługa wyjątku, ponieważ zależy ona od wielu czynników, takich jak algorytm alokacji pamięci na wyjątek czy ilość klauzuli