Projektowanie interfejsów C++

API (Application Programming Interface) to zagadnienie, z którym spotykamy się na co dzień. Czy jednak potrafimy dobrze zaprojektować interfejsy w języku C++? W jaki sposób C++ różni się od innych języków pod względem projektowania API? Czy rozumiemy semantykę różnych struktur tego języka? Lektura tego artykułu pozwoli czytelnikowi ugruntować wiedzę z zakresu tworzenia interfejsów w języku C++, a także zrozumieć, dlaczego biblioteka standardowa oraz boost w tak dużym stopniu korzystają z szablonów i tzw. metaprogramowania.

C++ A INNE JĘZYKI

Zanim zaczniemy temat projektowania interfejsów, należy przyjrzeć się samemu językowi C++. Przede wszystkim cieszy się on opinią szybkiego oraz, co ważniejsze, przewidywalnego pod względem wydajności i zajętości pamięci. Jest to ważne, ponieważ interfejsy, które piszemy, powinny umożliwiać implementację zgodną z założeniami języka. W końcu, jeżeli nasz interfejs będzie utrudniać stworzenie wydajnej implementacji, może się to skończyć tym, że nasza biblioteka nie będzie chętnie wykorzystywana. Mimo że wydaje się to oczywiste, nie jest regułą dla każdego języka czy biblioteki. Często języki są projektowane tak, żeby zapewnić maksymalny komfort pisania nowych funkcjonalności, pozostawiając kwestię wydajności na drugim miejscu.

Przede wszystkim należy mieć na uwadze, że C++ jako jeden z niewielu obecnie popularnych języków wyposażony jest w mechanizm „manualnego" zarządzania pamięcią. Z tego powodu nasze interfejsy będą różnić się od tych pisanych w językach z tzw. odśmiecaczem (ang. *garbage collector*). W interfejsach należy dużą uwagę zwrócić na tzw. własność (ang. *ownership*) obiektów. Interfejs klasy powinien bardzo precyzyjnie określać czas życia obiektów oraz to, kto jest ich właścicielem (kto jest odpowiedzialny za zwolnienie zasobu).

Interfejsy w C++ również będą wyglądać inaczej niż w innych językach, ponieważ nie ma w nim dedykowanych do tego mechanizmów. Dla porównania, w języku Rust mamy słowo kluczowe `trait`, a w Javie – `interface`; oba te słowa kluczowe służą do definiowania interfejsów. W C++ sprawa jest bardziej skomplikowana. Przede wszystkim mamy co najmniej 2 rodzaje interfejsów – statyczne oraz dynamiczne (możemy jeszcze mówić o interfejsach w metaprogramowaniu, ale to nie jest aż tak częsty przypadek dla normalnego użytkownika tego języka). Interfejsy dynamiczne zazwyczaj definiujemy jako klasy czysto abstrakcyjne, natomiast interfejsy statyczne definiujemy jako tzw. „type traity" lub od C++20 – koncepty. Co więcej, statycznych interfejsów możemy w ogóle nie definiować w samym języku, a jedynie jako umowny kontrakt.

Wszystkim tym zagadnieniom przyjrzymy się bliżej w dalszej części artykułu. Zacznijmy jednak od struktur języka i ich semantyki.

SEMANTYKA STRUKTUR JĘZYKA

Przyjrzyjmy się podstawowym mechanizmom wbudowanym w sam język C++ i temu, jak możemy je wykorzystać do projektowania API. Głównymi budulcami naszych bibliotek będą funkcje oraz typy. Zacznijmy od bliższego zapoznania się z funkcjami. Omówimy sobie ich rodzaje i to, do czego możemy ich użyć.

Funkcje

Funkcje w języku możemy podzielić na 3 podstawowe grupy: zwykłe funkcje, niestatyczne pola klasy oraz statyczne pola klasy (znane pod nazwami metody i metody statyczne). Każda grupa funkcji ma swoje zastosowania. Nie rozróżniam tutaj szablonowych wersji tych konstrukcji.

Funkcje zwykłe (freestanding functions)

Zwykłe funkcje zazwyczaj reprezentują proste algorytmy. W bibliotece standardowej możemy znaleźć całkiem sporą kolekcję takich funkcji, np. `std::swap` czy `std::abs`. Należą do nich także funkcje pomocnicze, które mają ukryć lub nazwać cechy języka, jak np. `std::move`, który ukrywa rzutowanie na rvalue referencję oraz `std::forward` enkapsulujący własności uniwersalnych referencji i tzw. mechanizm *reference collapsing*. Innym przykładem takich funkcji są operatory, np. `filesystem::operator/(const filesystem::path&, const filesystem::path&)`. Znajdziemy również tutaj funkcje, które są fabrykami: `std::make_unique` czy `std::make_shared`. Funkcje również pełnią rolę rozszerzania interfejsów klas. Algorytmy, które nie wymagają dostępu do prywatnych pól klasy, są zaimplementowane właśnie jako wolne funkcje (chociażby `std::erase_if`).

Funkcje statyczne

Funkcje statyczne są o wiele mniej użyteczne w języku C++ niż mogłoby się wydawać. O ile funkcje statyczne w niektórych językach jak np. Java są jedyną możliwością stworzenia funkcji, które nie wymagają obiektu do ich użycia, tak C++ ma od tego zwykłe funkcje. Do czego możemy użyć takich funkcji statycznych?

Głównym użyciem statycznych funkcji klasy jest tworzenie interfejsów czasu kompilacji. Dla przykładu wyobraźmy sobie funkcję:

Listing 1. Wybór algorytmu w czasie kompilacji

```
template <bool condition, typename U, typename V>
decltype(auto) if_else(){
    if constexpr (condition)
        return U::calculate();
    else
        return V::calculate();
}
```

W tym przypadku, ponieważ typy `U` oraz `V` mają zgodne interfejsy, możemy w trakcie kompilacji zdecydować, która funkcja ma zostać wykonana.

Innym częstym przypadkiem użycia statycznych funkcji jest implementacja fabryk. Dla przykładu przywołam tutaj kawałek kodu z biblioteki cppcoro:

Listing 2. Fabryki tworzące obiekty socket w bibliotece cppcoro

```
class socket {
public:
    static socket create_tcpv4(io_service& ioSvc);
    static socket create_tcpv6(io_service& ioSvc);
    static socket create_udpv4(io_service& ioSvc);
    static socket create_udpv6(io_service& ioSvc);
    //…
```

W klasie `socket` znajdują się 4 różne statyczne funkcje `create`. Powodem, dla którego nie został tutaj użyty konstruktor, jest fakt, że konstruktory nie mogą mieć różnych nazw, a parametry każdej z funkcji są takie same. Nie moglibyśmy więc w analogiczny sposób zaimplementować tego jako konstruktory. Pomimo tego zastosowanie funkcji `static` nie jest tutaj konieczne. Po pierwsze, fabryki moglibyśmy wyciągnąć poza klasę `socket`. Drugą opcją, preferowaną przeze mnie, byłoby mimo wszystko stworzenie konstruktorów z dodatkowym parametrem, który pozwalałby na rozróżnienie ich wywołania Na przykład:

Listing 3. Alternatywne podejście do tworzenia obiektów typu socket na bazie tzw. tagów

```
struct tcpv4_t{}tcpv4_tag;
struct tcpv6_t{}tcpv6_tag;
struct udpv4_t{}udpv4_tag;
struct udpv6_t{}udpv6_tag;

class socket {
public:
    socket(tcpv4_t, io_service& ioSvc);
    socket(tcpv6_t, io_service& ioSvc);
    socket(udpv4_t, io_service& ioSvc);
    socket(udpv6_t, io_service& ioSvc);
```

Przykład wywołania takiego konstruktora zaprezentowano w Listingu 4.

Listing 4. Wywołanie konstruktora zmodyfikowanej klasy socket

```
socket mySocket(tcpv4_tag, ioService);
```

Jakie są zalety takiego podejścia? Przede wszystkim w ten sposób napisana klasa łatwo komponuje się z innymi interfejsami. Zwróćmy uwagę, że większość algorytmów i typów zakłada, że korzystamy właśnie z konstruktorów do tworzenia obiektów. Dla przykładu, tak utworzona funkcja dobrze będzie komponować się nie tylko z funkcjami `std::make_unique` czy `std::make_shared`, ale także np. `std::vector::emplace_back`:

Listing 5. Wykorzystanie konstruktorów typu socket wraz z make_unique

```
std::make_unique<socket> (tcpv4_tag, ioService);
```

Analogiczne wywołanie wykorzystujące funkcje statyczne lub funkcje zaprzyjaźnione nie byłoby możliwe. Funkcje typu `std::make_unique` po prostu zakładają korzystanie z mechanizmów języka zgodnie z ich przeznaczeniem, co jest zaletą samą w sobie.

Używanie struktur języka zgodnie z ich przeznaczeniem zwiększa czytelność kodu – do konstruowania obiektów zostały przeznaczone konstruktory i to właśnie ich używamy do tworzenia obiektów. Z takim podejściem bez problemu odnajdziemy w kodzie logikę tworzenia obiektów – intuicyjnie będziemy ich przecież szukać właśnie w konstruktorze.

Nie oznacza to oczywiście, że biblioteka cppcoro jest ogólnie źle zaprojektowana czy w jakimkolwiek stopniu nieużywalna. Podany przykład miał jedynie pokazać, w jaki sposób wykorzystać mechanizmy C++, aby zdefiniować interfejs jak najbardziej przyjazny użytkownikowi.

Pomimo tego, że funkcje statyczne można zastąpić zwykłą funkcją, wiele osób nadal uważa, że mają swoje zastosowania. Na przykład John Lakos, członek komitetu standaryzacyjnego, twórca zasady Lakosa (więcej o niej w dalszej części artykułu) oraz twórca książki „Large Scale C++", uważa, że statyczne funkcje jak najbardziej mają sens przy tworzeniu tzw. klas pomocniczych (ang. *utility classes*). Taka klasa miałaby się składać z samych statycznych funkcji, które operują na innej klasie. Dla przykładu:

Listing 6. Przykład klasy pomocniczej

```
struct stringUtils{
    static std::vector<std::string_view> split(
                                std::string_view view,
                                std::string_view delimiter);
    static bool contains(std::string_view source,
                         std::string_view contained);
    //...
};
```

Jakie korzyści uzyskalibysmy z takiej implementacji? John Lakos podczas swojego wystąpienia na konferencji Code::Dive wspomniał o 7 powodach, wymieniając jednak tylko część z nich:

1. Funkcje te nie mogą być wybrane poprzez mechanizm ADL.
2. Zamykamy zbiór funkcji w jednej klasie oraz jednym pliku.

Osobiście zgadzam się z tymi argumentami, jednak nie są one dla mnie wystarczające. Fakt bycia niewybieranym przez mechanizm ADL możemy osiągnąć na kilka sposobów, np. definiując funktory. Dodatkowo, jeżeli zamiast struktury `stringUtils` stworzymy przestrzeń nazw o tej samej nazwie i nie zdefiniujemy w niej żadnych typów (czyli w zasadzie podmienimy łańcuch `struct` na `namespace`), to nasze funkcje również nie będą brały udziały w ADL, ponieważ żadne z argumentów nie będą pochodzić z tej przestrzeni nazw.

Jeżeli chodzi o zamknięcie zbioru funkcji w jednym pliku, to również nie zawsze to jest coś, co chcemy osiągnąć. Co, jeśli chcielibyśmy rozszerzyć naszą funkcjonalność? W podejściu ze statycznymi funkcjami musielibyśmy dziedziczyć po w zasadzie pustej strukturze lub tworzyć nową klasę z inną (niekoniecznie wiele znaczącą) nazwą. Nie uważam, że jest to wygodne rozwiązanie. W przypadku funkcji i przestrzeni nazw zawsze możemy rozszerzyć funkcjonalność poprzez otworzenie przestrzeni nazw i dodanie kolejnych funkcji.

Do reszty argumentów nie mogę się odnieść, ponieważ nie padły one podczas wystąpienia na wspomnianej na konferencji i ich po prostu nie znam.

Metody – funkcje jako niestatyczne pola klasy

Funkcje definiują zachowanie obiektów, pilnują tzw. niezmienników klasy (ang. *invariants*) i powinny stanowić minimalny zbiór