Czym zachwyci C++26

W momencie pisania tego artykułu (listopad 2025) C++26 znajduje się w stanie feature freeze od czerwca 2025 roku. Oznacza to, że nie zostaną już dodane żadne nowe funkcjonalności, a komitet skupia się na rozwiązywaniu uwag krajowych ciał normalizacyjnych oraz naprawie błędów. Finalizacja standardu przewidywana jest na marzec 2026 roku w Londynie. C++26 przynosi przełomowe zmiany, szczególnie w zakresie refleksji. Jest ona wg Herba Suttera „bardziej transformacyjna niż jakiekolwiek 10 innych głównych funkcjonalności, na które kiedykolwiek zagłosowaliśmy do standardu, razem wziętych".

ZMIANY W JĘZYKU

C++26 wprowadza kilkadziesiąt potwierdzonych zmian w języku, z których najważniejsze to refleksja (ang. *reflection*) w czasie kompilacji, kontrakty (po pięcioletniej przerwie) oraz szereg ulepszeń bezpieczeństwa eliminujących niezdefiniowane zachowanie (ang. *undefined behavior*, UB).

Refleksja – nowa era metaprogramowania

Refleksja – mechanizm języka pozwalający programowi analizować własne struktury, takie jak typy, pola czy funkcje, i wykorzystywać te informacje na etapie kompilacji do generowania lub modyfikowania kodu. Dzięki niej kompilator zyskuje możliwość „zaglądania" do programu i operowania na jego metadanych w kontrolowany, bezpieczny sposób.

Refleksja (P2996R13 wraz z sześcioma towarzyszącymi propozycjami) to w mojej opinii najważniejsza funkcjonalność dodana do C++ od wprowadzenia C++11. Operator refleksji ^^ (nazywany „unibrow operator") pozwala na refleksję nad typami, funkcjami i innymi elementami programu w czasie kompilacji. Odzwierciedlony element staje się wartością typu `std::meta::info`, którą można badać i manipulować. Drugi kluczowy operator – splice `[::]` – pozwala na „wklejenie" odzwierciedlonej informacji z powrotem do kodu, umożliwiając generowanie kodu na podstawie metadanych.

W Listingach 0 i 2 przedstawiłem przykładowe użycie refleksji w wersji zaimplementowanej przez *fork* kompilatora Clang od firmy Bloomberg [0], na tapet biorąc popularne bolączki programistów C++.

Kluczowe elementy używane w przykładach:

- ^^ (operator refleksji) – przekształca typ lub element w wartość `std::meta::info`
- `[::]` (splice operator) – „wkleja" metadane z powrotem do kodu, np. `[:member:]` rozwija się do rzeczywistej nazwy składowej
- `std::define_static_array()` – konwertuje sekwencję `meta::info` na statyczną tablicę, umożliwiając użycie w template for
- `template for` – pętla wykonywana w czasie kompilacji, iterująca po metadanych

W Listingu 0 przedstawiono zamianę wartości enumeracyjnych na wartości tekstowe odpowiadające ich nazwom, co dotychczas było możliwe tylko przy zastosowaniu hacków (jak np. `magic_enum`) lub dodatkowych programów wywoływanych przed kompilacją. W Listingu 2 natomiast przedstawiono serializację klas bez ingerencji w nie, co dotychczas wymagało użycia niestandardowego oprogramowania do parsowania/generowania kodu lub utworzenia go ręcznie.

Listing 0. Implementacja funkcji enum_to_string

```
#include <meta>
#include <print>
#include <string>
#include <type_traits>

enum class HttpStatus {
    OK = 200,
    NotFound = 404,
    InternalError = 500
};

// Convert any enum value to its string name at compile-time
template <typename E>
    requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value)
{
    std::string result = "<unknown>";

    // Reflect on the enum type using ^^ operator
    // Get all enumerators and convert to static array for
    // constexpr iteration
    template for (constexpr auto e :
                  std::define_static_array(
                      std::meta::enumerators_of(^^E)))
    {
        // Splice the enumerator value using [:e:]
        if (value == [:e:])
        {
            // Get the identifier name of the enumerator
            result =
            std::to_string(std::meta::identifier_of(e));
        }
    }
    return result;
}

int main() {
    // Compile-time usage
    constexpr auto ok_name = enum_to_string(HttpStatus::OK);
```

```
    static_assert(ok_name == "OK");
    auto print_status = [](HttpStatus status) {
        std::println("Status {}: {}",
            std::underlying_type_t<HttpStatus>(status),
            enum_to_string(status));
    };
    // Runtime usage
    print_status(HttpStatus::OK);
    print_status(HttpStatus::NotFound);
    print_status(HttpStatus::InternalError);

    // Unknown value
    print_status(HttpStatus(999));
}
```

Listing 1. Wykonanie kodu z Listingu 0

```
% ./reflection1
Status 200: OK
Status 404: NotFound
Status 500: InternalError
Status 999: <unknown>
```

Listing 2. Funkcja debug_print serializująca klasy

```
#include <meta>
#include <print>
#include <string>

struct Person {
    std::string name;
    int age;
    std::string email;
};

// Generic debug printer for any struct
// Prints all member names and their values
template <typename T>
void debug_print(const T& obj)
{
    std::println("{}:", std::meta::identifier_of(^^T));

    // Iterate over all non-static data members
    template for (constexpr auto member :
             std::define_static_array(
              std::meta::nonstatic_data_members_of(
                ^^T,
                std::meta::access_context::current())))
    {

        // Print member name and value using splice operator
        std::println("  {}: {}",
            std::meta::identifier_of(member),
            obj.[:member:]);
    }
}

int main() {
    Person alice {
        .name = "Alice Smith",
        .age = 30,
        .email = "alice@example.com"
    };

    Person bob {
        .name = "Bob Jones",
        .age = 25,
        .email = "bob@example.com"
    };

    debug_print(alice);
    std::println("");
    debug_print(bob);
}
```

Listing 3. Wykonanie kodu z Listingu 2

```
% ./reflection2
Person:
  name: Alice Smith
  age: 30
  email: alice@example.com

Person:
  name: Bob Jones
  age: 25
  email: bob@example.com
```

Uwaga: przykłady z Listingów 0 i 2 były kompilowane w oparciu o fork kompilatora Clang od firmy Bloomberg [0]. Kod zgodny z ostateczną wersją standardu C++26 może się nieznacznie różnić.

Szersze możliwości refleksji

Przedstawione przykłady to zaledwie wierzchołek góry lodowej. Refleksja w C++26 umożliwi scenariusze, które dotychczas wymagały zewnętrznych generatorów kodu lub były niemożliwe:

- Automatyczna serializacja do dowolnych formatów – wystarczy napisać jeden generyczny serializer, który obsłuży każdą strukturę, włączając w to zagnieżdżone kontenery STL, smart pointery i typy opcjonalne.
- ORM-y i mappery baz danych – automatyczne mapowanie struktur C++ na tabele SQL bez makr i zewnętrznych narzędzi.
- Dependency injection – kontenery DI mogące automatycznie wykrywać i wstrzykiwać zależności na podstawie typów parametrów konstruktora.
- Automatyczne generowanie GUI – tworzenie formularzy i interfejsów na podstawie struktury klas.
- Mock-i do testów – generowanie mock obiektów w czasie kompilacji bez frameworków typu GMock.
- Walidacja i sanityzacja – automatyczne sprawdzanie ograniczeń na polach struktur bez boilerplate'u.

Co kluczowe, wszystko to dzieje się w czasie kompilacji – zero narzutu w *runtime* (względem rozwiązań już istniejących), pełne bezpieczeństwo typów, możliwość optymalizacji przez kompilator.

Kontrakty - powrót po redesignie

Kontrakty (P2900R14) wracają do C++ po usunięciu z draftu C++20. Wprowadzają *preconditions* (`pre`), *postconditions* (`post`) i *assertions* (`contract_assert`) do specyfikacji interfejsów funkcji.

Standard definiuje trzy semantyki kontraktów:

- *ignore* – kontrakty są całkowicie pomijane, brak sprawdzania
- *observe* – naruszenie kontraktu wywołuje handler, ale wykonanie może być kontynuowane
- *enforce* – naruszenie kontraktu zawsze kończy program

Wybór semantyki jest *implementation-defined* – to implementacja (kompilator) decyduje, które tryby udostępnia i jak je kontrolować.

Typowo będzie to opcja kompilacji, np. `-fcontract-evaluation-semantic=observe` lub `-fcontract-evaluation-semantic=enforce`.

W kodzie z Listingu 4 znajdują się dwa naruszenia kontraktu funkcji `divide()`:

1. Dzielenie przez -1: jest to błąd logiczny, ponieważ tak go zdefiniowaliśmy. Zachowanie jest zdefiniowane dla prawie wszystkich wartości `int` – w szczególności `10`.
2. Dzielenie przez 0: undefined behavior.

W zależności od tego, czy użyjemy semantyki *enforce* czy *observe*, program zostanie przerwany po pierwszym naruszeniu (ang. *enforce*) albo zostanie tylko pokazana informacja diagnostyczna (ang. *observe*), a program będzie kontynuował wykonanie. W moim przypadku dalsze wykonanie sprowadziło się do zaraportowania informacji o kolejnym naruszeniu kontraktu (dzielenie przez 0) oraz zakończeniu działania programu, co jest jednym z nieskończenie wielu prawidłowych zachowań w przypadku UB.

Listing 4. Przykład użycia kontraktów

```
#include <print>

int divide(const int a, const int b)
    pre (b != 0 && b != -1)
    post (r: a == b * r + a % b)
{
    contract_assert(a >= 0 || b != 0);
    return a / b;
}

int main()
{
    // no contract violation
    std::println("10 / 3 = {}", divide(10, 3));
    // contract violation: b == -1
    std::println("10 / -1 = {}", divide(10, -1));
    // contract violation: b == 0
    std::println("10 / 0 = {}", divide(10, 0));
}
```

Listing 5. Wykonanie kodu z Listingu 4, semantyka observe

```
% gcc-cpp26-contracts/bin/g++ -std=c++26 \
    -fcontracts \
    -fcontracts-nonattr \
    -Wall \
    -Wextra \
    -pedantic \
    -fcontract-evaluation-semantic=observe \
    -o contracts1-observe \
    main.cpp

% ./contracts1-observe
10 / 3 = 3
contract violation in function int divide(int, int) at
                                main.cpp:7: b != 0 && b != -1
[assertion_kind: pre, semantic: observe,
                      mode: predicate_false, terminating: no]
10 / -1 = -10
contract violation in function int divide(int, int) at
                                main.cpp:7: b != 0 && b != -1
[assertion_kind: pre, semantic: observe,
                      mode: predicate_false, terminating: no]
zsh: floating point exception (core dumped)
                                          ./contracts1-observe
```

Listing 6. Wykonanie kodu z Listingu 4, semantyka enforce

```
% gcc-cpp26-contracts/bin/g++ -std=c++26 \
    -fcontracts \
    -fcontracts-nonattr \
    -Wall \
    -Wextra \
    -pedantic \
    -fcontract-evaluation-semantic=enforce \
    -o contracts1-enforce \
    main.cpp

% ./contracts1-enforce
10 / 3 = 3
contract violation in function int divide(int, int) at
                                main.cpp:7: b != 0 && b != -1
[assertion_kind: pre, semantic: enforce,
                      mode: predicate_false, terminating: yes]
terminate called without an active exception
zsh: IOT instruction (core dumped)  ./contracts1-enforce
```

Kontrakty nie są jeszcze dostępne w głównych gałęziach kompilatorów Clang i GCC, jednak są one aktywnie rozwijane. Eksperymentalne implementacje [1] pokazują, że komunikaty błędów będą znacznie bardziej informatywne niż tradycyjne asserty, włączając w to wartości zmiennych i pełny kontekst naruszenia. Komitet pracuje również nad semantyką *must enforce* dla krytycznych kontraktów, które muszą być sprawdzane nawet przy agresywnej optymalizacji.

Pack indexing – bezpośredni dostęp do parameter packów

Pack indexing (P2662R3) umożliwia bezpośredni dostęp do elementów parameter packów bez skomplikowanego metaprogramowania szablonowego.

Listing 7. Przykład pack indexing

```
#include <print>
#include <type_traits>

template<typename... Ts>
auto get_first(Ts... values)
{
    return values...[0];
}

template<typename... Ts>
auto get_last(Ts... values)
{
    return values...[sizeof...(values) - 1];
}

int main()
{
    std::println("First: {}", get_first(1, 2, 3, 4));
    std::println("Last: {}", get_last(1, 2, 3, 4));
}
```

Listing 8. Wykonanie kodu z Listingu 7

```
% ./pack_indexing
First: 1
Last: 4
```

ŚWIT POLSKIEGO IT · ARCHITEKT W KRAINIE ITIES

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

6/2025 (121)

Cena 32.90 zł (w tym VAT 8%)

CZYM ZACHWYCI C++ 26

PyTorch w NLP. Od tokenizacji do predykcji kolejnego słow

Wizualizacja kąta obrotu obiektu z wykorzystanie

Koniec bańki generatywnego AI

Nie żyje król, niech żyje król! O alternatywach go AutoMappera

Nowy numer już w empikach