C++20 – trzęsienie ziemi na koniec dekady

W sobotę, 15 lutego 2020, zakończyło się posiedzenie komisji standaryzacyjnej C++. Chwilę potem członkowie komisji poinformowali w serwisie Reddit [0], a także za pomocą innych kanałów komunikacji, że ostateczna wersja (szkicu) standardu (DIS – ang. Draft International Standard) jest gotowa do wysłania do tzw. National Bodies w celu poddania ostatecznemu głosowaniu, które to powinno być wyłącznie formalnością.

Cóż za zwieńczenie dekady[1]! C++20 – ponieważ tak będzie się nazywał nowy standard, jeśli wejdzie w życie w 2020 roku – będzie zawierał szereg rewolucyjnych i od dawna wyczekiwanych zmian. Nawet czytając same nagłówki sekcji tego artykułu, można zauważyć, że C++20 będzie wszystkim tym, czym miał być C++17 – i więcej! W opinii autora nadchodząca iteracja standardu wprowadza większą rewolucję niż osławiony C++11. Nie obejdzie się jednak bez łyżki dziegciu.

MODUŁY

Bez wątpienia jedną z najważniejszych zmian jest umożliwienie odejścia od archaicznego modelu dołączania zewnętrznego kodu poprzez dyrektywę preprocesora `#include`. Zamiast tego używane będą słowa `module` i `import`. Słowo `module` definiuje moduł, a `import` to instrukcja załączająca moduł.

Warto tutaj zaznaczyć, że nie są to nowe słowa kluczowe, tylko identyfikatory ze specjalnym znaczeniem (tak jak `override` i `final`). Dzięki temu zachowana zostaje kompatybilność z poprzednimi standardami, gdzie kod z Listingu 0 jest w pełni poprawny i taki pozostaje:

Listing 0. Poprawny kod C++17 pozostaje poprawny w C++20 [1]

```
class module{};

int main()
{
    module m;
}
```

Nie wszędzie jednak udało się zachować pełną kompatybilność wsteczną: Listingi 1 i 2 pokazują na przykładzie zmiennych globalnych kod, który wraz z C++20 przestaje się kompilować:

Listing 1. Poprawny kod C++17, niepoprawny w C++20 [2]

```
class module{};

module m;

int main()
{
}
```

Listing 2. Poprawny kod C++17, niepoprawny C++20 (w momencie pisania tego artykułu przyjmowany przez clang i gcc)

```
class module{};

namespace foo
{
    module m;
}

int main()
{
}
```

Choć z powyższego opisu mogłoby się wydawać, że jest to tylko formalna zmiana, to należy podkreślić, że odejście od preprocesora jest znaczącą innowacją. Preprocesor jest dziedzictwem języka C, a jego *design* pamięta lata 70. poprzedniego stulecia. Jego zasada działania jest bardzo prosta, ale jednocześnie problematyczna: dyrektywa `include` podmienia podaną nazwę lub ścieżkę do pliku na jego zawartość. Można to zaobserwować, dodając przełącznik `-E` do polecenia wywołania kompilatora gcc lub clang:

Listing 3. Zasada działania preprocesora

```
# cat x.cpp
#include "test"

int main()
{
        return foo();
}

# cat test
int foo()
{
        return 42;
}

# g++ x.cpp -E
# 1 "x.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "x.cpp"
# 1 "test" 1

int foo()
{
 return 42;
}
# 2 "x.cpp" 2

int main()
{
 return foo();
}
```

1. Dekadą popularnie nazywa się okres od roku zakończonego 1 do roku zakończonego 0. Jest tak, ponieważ daty liczone są od roku zakończonego 1, a nie 0 (*notabene*, koncept zera dotarł do Europy dopiero w poprzednim millenium), tak więc pierwsza dekada to były lata [1–10], pierwsze stulecie to lata [1–100], a pierwsze millenium to lata [1–1000].

Preprocesor to rozwiązanie o dekady starsze od przestrzeni nazw, przeładowywania funkcji, klas i szablonów. Z tego powodu nie współgra on z nimi w sposób naturalny. Najczęściej wymieniane problemy to:

» Kolizje nazw – `windows.h` deklaruje makro `max`, które uniemożliwia korzystanie z `std::max()` (Listing 4).
» Utrudnienie osobnej kompilacji – zmiana w dowolnym z nagłówków propagowana jest kaskadowo do każdej jednostki kompilacji (w uproszczeniu: pliku .cpp), która z niego korzysta bezpośrednio lub pośrednio. Znaczącym problemem jest też prekompilacja nagłówków.
» Zależność od kolejności użycia – twórcy bibliotek muszą podejmować dodatkowe kroki, aby zagwarantować poprawność użycia ich kodu niezależnie od kolejności dyrektyw `#include`. Jeśli tak się nie stanie, to kod może się nie skompilować, albo, co gorsze, skompilować z inną semantyką. Za przykład może służyć `windows.h` i `winsock2.h` – jeśli `winsock2.h` nie jest dodane przed `windows.h`, to `windows.h` zostanie użyte bez zmiennej `_WINSOCKAPI_`.
» Brak odporności na powtórne użycie – kilkukrotne wywołanie dyrektywy `#include` dla tego samego pliku spowoduje kilkukrotne jego dodanie do kodu wynikowego. W przypadku większych projektów, gdzie różne moduły wzajemnie od siebie zależą, jest to właściwie nieuniknione (np. `main.cpp` używa `<iostream>` oraz `<myclass>`, a `<myclass>` również załącza `<iostream>`). Aby się przed tym bronić, trzeba korzystać z obejść takich jak `#pragma once` lub *include guardy*.

Listing 4. Przykład kolizji nazw spowodowany załączonym nagłówkiem windows.h

```
#include <algorithm>
#include <windows.h>

int main()
{
    return std::max(42, 0);
}
```

Listing 5. Błąd zwrócony przez kompilator

```
1>Main.cpp
1> main.cpp(6): error C2589: '(': illegal token on right side of '::'
1> main.cpp(6): error C2062: type 'unknown-type' unexpected
1> main.cpp(6): error C2059: syntax error: ')'
1>Done building project "Test.vcxproj" -- FAILED.
```

Rozwiązaniem tych problemów jest system modułów. Moduły podlegają osobnej kompilacji i nie powodują zanieczyszczeń przestrzeni nazw (ang. *namespace pollution*). Dzięki temu, że zależne moduły nie wpływają na siebie nawzajem, kolejność ich dołączania jest kompletnie bez znaczenia. Możliwa jest też znacznie bardziej rozszerzona prekompilacja.

Niestety, na chwilę pisania tego artykułu żaden kompilator nie wspierał kompilacji z użyciem modułów (clang od wersji 8 sugeruje częściowe wsparcie, ale w opinii autora nie jest to nawet wersja alpha). Dlatego też autor posłuży się przykładem z artykułu (Programista 2/2019), opartego o GCC 6 z Modules TS [3]:

Listing 6. Przykładowy kod wykorzystujący moduły – wywołanie [3]

```
> ./gccmod/bin/g++ -fmodules-ts main.cpp mod.cpp -o mod
> ./mod
Hello, Modular World!
The answer is 42
```

Listing 7. Przykładowy kod wykorzystujący moduły – plik mod.cpp [3]

```
module;
export module mod;

#define ANSWER 42

export int answer()
{
    return ANSWER;
}
```

Listing 8. Przykładowy kod wykorzystujący moduły – plik main.cpp [3]

```
#include <iostream>

import mod;

int main()
{
    std::cout << "Hello, Modular World!\n";
    std::cout << "The answer is " << answer() << '\n';
}
```

W ramach ciekawostki można dodać, że pierwsze zachowane propozycje dodania modułów do C++ pojawiły się już w 2004 roku w postaci dokumentu N1736, którego autorem był Daveed Vandervoorde [4].

KONCEPTY

Koncepty (ang. *concepts*) są kolejną ogromną zmianą w języku. Są też kolejną zmianą, która na horyzoncie była od dekad – pierwsze formalne propozycje standaryzacji pojawiły się już w 2003 roku. To próby włączenia, między innymi, właśnie konceptów spowodowały opóźnienie w tworzeniu standardu o nazwie roboczej C++0x, znanego później jako C++11. W momencie wydania C++11 koncepty były zapowiedziane jako element C++17. Lament autora tego artykułu spowodowany ich brakiem można zobaczyć w artykule z numeru 10/2016 [5].

Koncepty dodają do standardu nowe *słowa kluczowe*: `concept` i `requires`. `concept` jest używane do definicji nowych konceptów, a `requires` jest słowem „przeładowanym" (podobnie jak `noexcept`): służy zarówno do oznaczania wymagań funkcji, jak i do ich definicji.

Dodanie nowych słów kluczowych oznacza, że kod z Listingu 9 się nie skompiluje w C++20:

Listing 9. Poprawny kod C++17, błędny w C++20 [6]

```
int main()
{
    int requires;
}
```

Listing A. Listing z błędem w wywołaniu funkcji print_if()

```
#include <iostream>
#include <iterator>
#include <vector>

template<typename R, typename P>
void print_if(R&& r, P&& p) {
    for(auto const& el : r) {
        if(p(el))
            std::cout << el << ", ";
    }
    std::cout << '\n';
}

int main()
{
    std::vector a{1,2,3,4,5};
    print_if([](int x){ return x % 2 == 0; }, a);
}
```

Podstawowym celem wprowadzenia konceptów jest zwiększenie czytelności kodu. Weźmy za przykład kod z Listingu A, gdzie z powodu drobnej pomyłki programista zamienił kolejnością argumenty funkcji `print_if()`.

Próba kompilacji tego programu na komputerze autora spotkała się z takim komunikatem błędu:

Listing B. Kompilacja kodu z Listingu A

```
g++ -g -std=c++2a wb.cpp -o wb
wb.cpp: In instantiation of 'void print_if(R&&, P&&) [with R =
main()::<lambda(int)>; P = std::vector<int>&]':
wb.cpp:19:48:   required from here
wb.cpp:9:5: error: 'begin' was not declared in this scope; did
you mean 'std::begin'?
    9 |     for(auto const& el : r) {
      |     ^~~
      |     std::begin
In file included from /usr/include/c++/9.3.0/string:54,
                 from /usr/include/c++/9.3.0/stdexcept:39,
                 from /usr/include/c++/9.3.0/array:39,
                 from /usr/include/c++/9.3.0/tuple:39,
                 from /usr/include/c++/9.3.0/functional:54,
                 from /usr/include/c++/9.3.0/pstl/
glue_algorithm_defs.h:13,
                 from /usr/include/c++/9.3.0/algorithm:71,
                 from all.hpp:10,
                 from wb.cpp:1:
/usr/include/c++/9.3.0/bits/range_access.h:105:37: note:
'std::begin' declared here
  105 |   template<typename _Tp> const _Tp* begin(const
valarray<_Tp>&);
      |                                     ^~~~~
wb.cpp:9:5: error: 'end' was not declared in this scope; did you
mean 'std::end'?
    9 |     for(auto const& el : r) {
      |     ^~~
      |     std::end
In file included from /usr/include/c++/9.3.0/string:54,
                 from /usr/include/c++/9.3.0/stdexcept:39,
                 from /usr/include/c++/9.3.0/array:39,
                 from /usr/include/c++/9.3.0/tuple:39,
                 from /usr/include/c++/9.3.0/functional:54,
                 from /usr/include/c++/9.3.0/pstl/
glue_algorithm_defs.h:13,
                 from /usr/include/c++/9.3.0/algorithm:71,
                 from all.hpp:10,
                 from wb.cpp:1:
/usr/include/c++/9.3.0/bits/range_access.h:107:37: note:
'std::end' declared here
  107 |   template<typename _Tp> const _Tp* end(const
valarray<_Tp>&);
      |                                     ^~~
make: *** [Makefile:5: all] Error 1
The terminal process terminated with exit code: 2
```

Listing C. Wykorzystanie konceptów

```
#include <iostream>
#include <iterator>
#include <ranges>
#include <vector>

template<std::ranges::input_range R, std::unary_predicate P>
void print_if(R&& r, P&& p) {
    for(auto const& el : r) {
        if(p(el))
            std::cout << el << ", ";
    }
    std::cout << '\n';
}

int main()
{
    std::vector a{1,2,3,4,5};
    print_if([](int x){ return x % 2 == 0; }, a);
}
```

W opinii autora komunikat błędu jest mało czytelny. Dzieje się tak, ponieważ podczas konkretyzacji szablonów kompilator nie wie, co dany typ szablonowy ma reprezentować, i może zgłosić błąd dopiero znacznie dalej w stosie wywołań, w miejscu gdzie podstawienie typu się nie powiodło.

Z użyciem konceptów z założenia powinno być zupełnie inaczej. Na przykładzie Listingu C już w momencie próby dopasowania do szablonu wykryty zostanie błąd – wektor nie jest predykatem, a lambda-predykat nie jest zakresem (ang. *range*).

Niestety, na chwilę obecną żaden z kompilatorów nie umożliwia skompilowania tego kodu, choć GCC 10 i MSVC 19.23 dokumentują wsparcie językowe dla konceptów.

KORUTYNY

Korutyny (ang. *coroutines*) to realizacja pomysłu rozdzielenia stanu wykonania (np. zmiennych lokalnych funkcji) od wątku wykonania programu. Pozwala to na pisanie nieblokującego kodu z zachowaniem zwięzłości charakterystycznej dla kodu sekwencyjnego. W wielu językach programowania przeskoki między kontekstami są ukrywane przez słowa kluczowe `await` i `yield`. C++20 dodaje tutaj trzy nowe słowa kluczowe: `co_yield`, `co_return` i `co_await`. Ich nazwy są dość kontrowersyjne, ale trzeba zwrócić uwagę, że komisja standaryzacyjna w niektórych przypadkach stara się maksymalizować zachowanie kompatybilności wstecznej.

Listing D. Przykładowy kod używający korutyn [7] (źródło: propozycja zmiany standardu N4736 [8])

```
#include <iostream>
#include <coroutine>

struct generator {

    struct promise_type;

    using handle =
        std::coroutine_handle<promise_type>;

    struct promise_type {

        int current_value;

        static auto
        get_return_object_on_allocation_failure() {
            return generator{ nullptr };
        }

        auto get_return_object() {
            return generator{ handle::from_promise(*this) };
        }

        auto initial_suspend() {
            return std::experimental::suspend_always{};
        }

        auto final_suspend() {
            return std::experimental::suspend_always{};
        }

        void unhandled_exception() {
            std::terminate();
        }

        void return_void() {}

        auto yield_value(int value)
        {
            current_value = value;
            return std::suspend_always{};
        }
    };

    bool move_next() {
        return coro ? (coro.resume(), !coro.done()) : false;
    }

    int current_value() {
```

```
        return coro.promise().current_value;
    }

     generator(generator const&) = delete;

    generator(generator&& rhs)
        : coro(rhs.coro)
    {
        rhs.coro = nullptr;
    }

    ~generator()
    {
        if (coro)
            coro.destroy();
    }

private:
    generator(handle h)
        : coro(h)
    {
    }

    handle coro;
};

generator f()
{
    co_yield 1;
    co_yield 2;
}

int main()
{
    auto g = f();
    while (g.move_next())
        std::cout << g.current_value() << std::endl;
}
```

Więcej o korutynach można przeczytać w artykule Dawida Pilarskiego w numerze 2/2019 [3].

ZAKRESY (RANGES)

Operowanie na kolekcjach danych wyłącznie za pomocą biblioteki standardowej było w C++ dość uciążliwe – wymagało przekazywania par iteratorów dla każdej kolekcji, chociaż w znakomitej większości przypadków operuje się na wszystkich jej elementach.

Choć może wydawać się to problemem naciąganym, wcale tak nie jest: jako że każda wynikowa kolekcja musi posłużyć jako źródło dwóch argumentów (iteratorów), nie ma możliwości, aby była to zmienna anonimowa. Powoduje to zaśmiecenie funkcji nazwanymi zmiennymi tymczasowymi, co jednoznacznie przekłada się na utrudnienie szybkiego zrozumienia jej działania przez programistę.

Za przykład niech posłuży program obliczający kwotę wydawaną w firmie na pensje poniżej 10000 pieniędzy:

Listing E. Program z użyciem narzędzi C++17 [9]

```
// WYŁĄCZNIE DLA UPROSZCZENIA KODU W PRZYKŁADZIE
// Nie rób tego w domu! Ani w pracy!
// double to okropny typ danych do opisu pieniędzy
using money = double;

struct employee
{
    std::string name;
    money salary;
};

auto employees()
{
    return std::vector<employee>{
        { "Kazimierz Wielki", 5000 },
        { "Bolesław Chrobry", 2000 },
        { "Mieszko I", 1000 },
        { "Władysław Jagiełło", 10000 },
        { "Zygmunt I Stary", 20000 },
    };
}

int main()
{
    auto all_employees = employees();
    std::vector<employee> filtered;
    std::copy_if(
        all_employees.cbegin(),
        all_employees.cend(),
        std::back_inserter(filtered),
        [](auto const& e){ return e.salary < 10000; }
    );
    std::vector<money> salaries;
    std::transform(
        filtered.cbegin(),
        filtered.cend(),
        std::back_inserter(salaries),
        [](auto const& e){ return e.salary; }
    );
    auto sum = std::accumulate(
        salaries.cbegin(),
        salaries.cend(),
        money{}
    );
    std::cout << sum << '\n';
}
```

Jak łatwo zauważyć, aby policzyć sumę, potrzebnych było kilka zmiennych pomocniczych i dość sporo kodu. Ten sam kod, używający biblioteki ranges-v3 Erica Nieblera [A], która była prekursorem zmian w standardzie, wygląda następująco:

Listing F. Kod z Listingu E wykorzystujący range-v3 (definicję typów pominięto)

```
int main()
{
    using namespace ranges;
    auto all_employees = employees();

    auto less_than_10k = [](auto const& e) {
        return e.salary < 10000;
    };
    auto get_salary = [](auto const& e) {
        return e.salary;
    };

    auto sum = ranges::accumulate(
        all_employees |
            views::filter(less_than_10k) |
            views::transform(get_salary),
        0
    );
    std::cout << sum << '\n';
}
```

O ile nie jest to rozwiązanie idealne, to jednak znacząco czytelniej przedstawia wysokopoziomowo wykonane działania. Niestety, obecna biblioteka range-v3 jest dość powolna w kompilacji (więcej na ten temat w wydaniu 11/2018 [B]). Na tej podstawie można domniemywać, że również wersje z implementacji bibliotek standardowych będą obarczone znacznym narzutem – nie ma jednak pewności, ponieważ w chwili pisania tego artykułu żaden kompilator nie informował o wsparciu *ranges*.

STD::FORMAT

Adaptacja popularnej biblioteki fmt [C] ostatecznie trafiła do standardu. W chwili pisania artykułu żaden z kompilatorów jeszcze jej nie oferował. Będzie ona zadeklarowana w nagłówku `<format>`, a jej przykładowe użycie przedstawiono w Listingu 10.

Listing 10. Przykładowe użycie std::format

```
#include <format>

int main() {
    std::cout <<
        std::format("Hello, formatted {}!\n", "world");
}
```

OPERATOR<=>

`operator<=>`, popularnie zwany operatorem statku kosmicznego (ang. *the spaceship operator*), służy do porównania trójstronnego. Pozwala to uniknąć wielokrotnego wykonywania porównań, aby sprawdzić ekwiwalentność – np. szablon funkcji `std::sort()` lub `std::set` przyjmują komparator, domyślnie `std::less`. Aby za jego pomocą sprawdzić ekwiwalentność obiektów A i B, należy wykonać dwa porównania: $A = B \Leftrightarrow \neg(A < B) \wedge \neg(B < A)$. Wykorzystując `operator<=>`, porównanie byłoby tylko jedno.

`operator<=>` może zwrócić wartości następujących typów:

- `std::strong_ordering` – wszystkie wartości są porównywalne. Dwie wartości ekwiwalentne są nierozróżnialne (np. liczby całkowite). Eksponuje cztery stałe:
  - `less` – porównanie ma wartość „mniej”,
  - `equivalent`, `equal` – stałe o identycznej wartości, porównanie ma wartość „równe”,
  - `greater` – porównanie ma wartość „więcej”.
- `std::weak_ordering` – wszystkie wartości są porównywalne. Dwie ekwiwalentne wartości nie muszą być równe (np. porównanie odległości punktów od początku układu współrzędnych). Eksponuje trzy stałe:
  - `less` – porównanie ma wartość „mniej”,
  - `equivalent` – porównanie ma wartość „ekwiwalentne”,
  - `greater` – porównanie ma wartość „więcej”.
- `std::partial_ordering` – pozwala na nieporównywalne wartości (np. NaN). Dwie ekwiwalentne wartości nie muszą być równe. Eksponuje cztery stałe:
  - `less` – porównanie ma wartość „mniej”,
  - `equivalent` – porównanie ma wartość „ekwiwalentne”,
  - `greater` – porównanie ma wartość „więcej”,
  - `unordered` – porównanie ma wartość „nieporównywalne”.

Definicje tych klas znajdują się w nagłówku `<compare>`.

Listing 11. Operator<=> – przykładowe użycie [D]

```
#include <compare>

int main()
{
    auto i = 1 <=> 2;
    auto d = 1.0 <=> 2.0;
    auto du = 1.0 <=> sqrt(-1);

    static_assert(
        std::is_same_v<decltype(i), std::strong_ordering>
    );
    static_assert(
        std::is_same_v<decltype(d), std::partial_ordering>
    );
    static_assert(
        std::is_same_v<decltype(du), std::partial_ordering>
    );
    assert(i == std::strong_ordering::less);
    assert(d == std::partial_ordering::less);
    assert(du == std::partial_ordering::unordered);
}
```

DESIGNATED INITIALIZERS

Desygnowane inicjalizatory (ang. *designated initializers*) to rozszerzenie inicjalizacji wprowadzone do języka C w 1999 roku. Pozwala ono na użycie nazw zmiennych struktur oraz indeksów tablic podczas ich inicjalizacji, znacząco zwiększając czytelność kodu i uniezależniając go od ewentualnej zmiany definicji struktury. Przykładowy kod znajduje się w Listingu 12

Listing 12. Designated initializers w C [E]

```
#include <assert.h>

struct foo
{
    int a;
    double b;
};

int main()
{
    struct foo tmp = {
        .b = 13.37,
        .a = 42
    };
    int arr[100] = { [97] = 42, [1] = 8 };
    assert(tmp.a == 42);
    assert(tmp.b == 13.37);
    assert(arr[0] == 0);
    assert(arr[1] == 8);
    assert(arr[97] == 42);
}
```

Jak widać na przykładzie, w C kolejność zapisu inicjalizatorów nie ma znaczenia. W C++20 jest inaczej: inicjalizatory muszą być zapisane w kolejności ich definicji w ciele klasy i żaden nie może być pominięty. Jest to spowodowane tym, że w C++ tworzenie i niszczenie obiektów może mieć znaczące efekty uboczne, których kolejność musi zostać zachowana. Nie można też inicjalizować w ten sposób tablic – w takim przypadku wystąpiłaby kolizja syntaktyczna z wyrażeniem lambda.

Listing 13. Designated initalizers w C++20 [F]

```
#include <cassert>

class foo
{
public:
    int a;
    double b;
};

int main()
{
    struct foo tmp = {
        .a = 42,
        .b = 13.37
    };
    assert(tmp.a == 42);
    assert(tmp.b == 13.37);
}
```

ROZSZERZENIE CONSTEXPR

Od C++11 w każdym kolejnym wydaniu standardu programista ma coraz szersze możliwości wykonania kodu w czasie kompilacji. Istotne nowości w C++20 to:

- funkcje `consteval` – muszą one zostać wywołane z kontekstu wywołania `consteval` lub ich wynik musi być przypisany do stałej. Niestety ta definicja jest niedoskonała, co zostanie pokazane w dalszej części tej sekcji. `consteval` jest nowym słowem kluczowym.

- `std::is_constant_evaluated()` – magiczna funkcja pozwalająca zorientować się, czy właśnie wykonujemy kod podczas kompilacji, czy podczas normalnego działania programu.
- `constinit` – nowe słowo kluczowe. Oznacza, że dany obiekt o statycznym czasie życia musi mieć stałą wartość inicjalizującą.
- `constexpr std::vector`
- `constexpr try i catch`
- `constexpr unie`
- `constexpr dynamic_cast i typeid`

Listing 14. constinit w praktyce (źródło: [10])

```
const char *g() { return "dynamic initialization"; }
constexpr const char *f(bool p) { return p ? "constant
initializer" : g(); }

constinit const char *c = f(true); // OK.
constinit const char *d = f(false); // ill-formed
```

Listing 15. Przykładowy kod korzystający z nowych funkcjonalności

```
consteval std::vector<int> generate(int n)
{
    std::vector<int> ret;
    for(int i = 0; i < n; i++) {
        ret.push_back(i * i);
    }
    return ret;
}

int main()
{
    constexpr static auto vec = generate(5);
}
```

W Listingu 15 przedstawiona jest funkcja `consteval` zwracająca `constexpr std::vector` – niestety jeszcze żaden kompilator nie może pochwalić się implementacją pozwalającą skompilować ten kod.

W Listingu 16 przedstawione zostało wykorzystanie funkcji `std::is_constant_evaluated()`. Należy zauważyć, że *nie został w nim użyty constexpr if*. Jest to celowe działanie, ponieważ użycie `constexpr if` to subtelny błąd: `constexpr if` gwarantuje wykonanie w czasie kompilacji, więc funkcja `std::is_constant_evaluated()` będzie ewaluowana zawsze do wartości prawdziwej. Zmienna a będzie miała wartość 1, a zmienna b: 2.

Listing 16. Poprawne użycie funkcji std::is_constant_evaluated() [11]

```
constexpr int foo()
{
    if(std::is_constant_evaluated()) {
        return 1;
    } else {
        return 2;
    }
}

int main()
{
    constexpr auto a = foo();
    auto b = foo();
    DBG(a);
    DBG(b);
}
```

Niestety, dwie najciekawsze nowości – funkcje `consteval` i `std::is_constant_evaluated()` – zupełnie nie są ze sobą kompatybilne. Jak pokazują autorzy dokumentu P1938R0 [12], ponieważ `consteval` wymaga wywołania z innej funkcji `consteval` lub przypisania do stałej, poniższy kod jest nieprawidłowy.

Listing 17. Brak współpracy consteval i std::is_constant_evaluated() (źródło: [12])

```
consteval int f(int i) { return i; }

constexpr int g(int i) {
    if (std::is_constant_evaluated()) {
        return f(i) + 1; // <==
    } else {
        return 42;
    }
}

consteval int h(int i) {
    return f(i) + 1;
}
```

INNE

Feature test macros

Są to makra pozwalające na sprawdzanie obecności konkretnych funkcjonalności. Bez nich nie było możliwości przenośnego sprawdzenia, czy dana implementacja oferuje jakąś funkcjonalność.

Listing 18. Brak współpracy consteval i std::is_constant_evaluated() (źródło: [12])

```
#ifdef __has_cpp_attribute
#   if __has_cpp_attribute(noreturn)
#   else
static_assert(false, "Need noreturn");
#   endif
#endif

int main()
{
}
```

jthread

Klasa o zbliżonym zachowaniu do `std::thread`, ale z następującymi istotnymi różnicami:

- Destruktor automatycznie wywołuje `join()`, jeśli zachodzi taka potrzeba (czyli wątek jest w stanie *joinable*).
- Jeśli wykonywana funkcja przyjmuje `std::stop_token` jako pierwszy argument, można przekazać jej życzenie, aby się zakończyła.

Listing 19. Użycie std::jthread [13]

```
void good(std::stop_token st)
{
    while(st.stop_requested())
        std::this_thread::sleep_for(10ms);
}

int main()
{
    std::jthread t{good};
    std::this_thread::sleep_for(1s);
}
```

char_8t

Nowy, osobny typ służący do przechowywania danych w formacie UTF-8. Powstał po to, by można było specjalizować szablony i przeładowywać funkcje specjalnie dla tego standardu kodowania.

Nowe atrybuty

- `[[nodiscard]]` z wiadomością – dodana została możliwość dodania wiadomości do atrybutu.
- `[[no_unique_address]]` – niestatyczna zmienna klasy nie musi mieć unikalnego adresu, a co za tym idzie, może zajmować 0 bajtów pamięci. Jest to zalegitymizowanie oraz rozszerzenie techniki znanej jako *Empty Base Optimization*.
- `[[likely]]` i `[[unlikely]]` – atrybuty o nazewnictwie wzorowanym kernelem Linuksa. *Likely* informuje kompilator, że dane zdarzenie jest bardziej prawdopodobne, a *unlikely* – że wręcz przeciwnie. Pozwala to na optymalizację wyrażeń warunkowych i pętli. W opinii autora jest to nazewnictwo błędne: często zdarza się, że zdarzenie rzadkie powinno mieć optymalną ścieżkę wykonania, np. obsługa zawału w rozruszniku serca albo obsługa nadciągających rakiet balistycznych

std::span

Jest to nowy typ o funkcjonalności zbliżonej do `std::string_view` lub `gsl::array_view` z biblioteki GSL, czyli jest to prosty typ implementujący widok na tablicę. W przeciwieństwie do `std::string_view` może on wskazywać na tablicę dowolnego typu. Ponadto pozwala on na modyfikację danych w tablicy, do której się odnosi.

Listing 1A. Przykład użycia std::span [14]

```
#include <span>

int main()
{
    int arr[4]{};

    std::span s(arr);

    s[0] = 42;

    assert(arr[0] == 42);
}
```

Wyrażenia lambda

Lista parametrów szablonowych

Od C++20 można *explicite* wywołać listę parametrów szablonowych wyrażenia lambda – tak jak w typowym szablonie. Dzięki temu możliwe jest np. wygodniejsze użycie *perfect forwardingu*. Przykładowy kod z C++17 (Listing 1B) można zamienić na ten z Listingu 1C

Listing 1B. Lambda C++17 – uzyskanie typu parametru szablonowego wymaga pewnej gimnastyki

```
[](auto&& foo) {
    using T = decltype(foo);
    return std::forward<T>(foo);
};
```

Listing 1C. Lambda C++20 z listą parametrów szablonowych

```
[]<typename T>(T&& foo) {
    return std::forward<T>(foo);
};
```

Lambdy w nieewaluowanych kontekstach wywołania

Po angielsku: *lambdas in unevaluated contexts*. W poprzednich wersjach C++ wyrażenia lambda nie mogły się znajdować w kodzie,

który nie był wykonywany (np. parametr `sizeof`). Od C++20 to się zmienia, choć niestety nie dotyczy to kontekstu SFINAE.

Listing 1D. Przykład ze standardu (§ 13.10.2 [temp.deduct]/9)

```
template <class T>
  auto f(T) -> decltype([]() { T::invalid; } ());
void f(...);
f(0);
// error: invalid expression not part of the immediate context

template <class T, std::size_t = sizeof([]() { T::invalid; })>
  void g(T);
void g(...);
g(0);
// error: invalid expression not part of the immediate context

template <class T>
  auto h(T) -> decltype([x = T::invalid]() { });
void h(...);
h(0);
// error: invalid expression not part of the immediate context

template <class T>
  auto i(T) -> decltype([]() -> typename T::invalid { });
void i(...);
i(0);
// error: invalid expression not part of the immediate context

template <class T>
  auto j(T t) -> decltype(
      [](auto x) -> decltype(x.invalid) { } (t));   // #1
void j(...);                                        // #2
j(0);
// deduction fails on #1, calls #2
```

Domyślna konstrukcja lambd

Wyrażenia lambda bez stanu (ang. *stateless lambdas*) mogą od teraz być domyślnie konstruowane, co równa je w użyteczności ze standardowymi obiektami funkcyjnymi:

Listing 1E. Lambda bez stanu zostaje użyta w std::set

```
int main()
{
    using cmp = decltype([](int a, int b){ return a < b; });
    std::set<int, cmp> s{1,2,3};
}
```

Jawne łapanie this

Od najnowszego standardu możliwe jest łapanie `this` w *capture list* pomimo użycia = do kopiowania wszystkich użytych w danej lambdzie zmiennych. Według propozycji standaryzującej ma to na celu zapewnienie równowagi z łapaniem przez referencję.

Listing 1F. Jawne łapanie this

```
struct foo
{
    foo() {
        [=,this]{}; // OK (C++20)
        [&,this]{};
    }
};
```

PODSUMOWANIE

C++20 to olbrzymie trzęsienie ziemi w świecie C++. W tym artykule przedstawiono zaledwie część zmian, w opinii autora tę najistotniejszą. Nadchodzący standard niewątpliwie będzie kolejnym epokowym wydarzeniem na miarę C++11. Niemniej jednak można zauważyć pewien pośpiech i brak koordynacji komisji standaryzacyjnej:

- Z jednej strony w konceptach dodawane są nowe słowa kluczowe łamiące kompatybilność – np `requires` – a z drugiej korutyny otrzymują ohydnie wyglądające potworki takie jak `co_await`,
- Kontrakty zostały przesunięte „na później”,
- `std::is_constant_evaluated()` nie współgra z `consteval`. W opinii autora jest to przeoczenie na miarę braku `std::make_unique()` w C++11,
- Jedyne dostępne implementacje *ranges* są na razie zabójcze dla kompilatorów i narzędzi do pracy z kodem (modele kodu, IDE).

Pomimo tego autor wyczekuje momentu, gdy C++20 będzie w pełni dostępny w jego ulubionych kompilatorach.

Bibliografia

[0] https://www.reddit.com/r/cpp/comments/f47x4o/202002_prague_iso_c_committee_trip_report_c20_is/
[1] https://wandbox.org/permlink/mKa7BhF3GwGW8w0G
[2] https://wandbox.org/permlink/gGKl9aYXFZ3pnrCi
[3] https://programistamag.pl/c20-%E2%80%8B-%E2%80%8B-kolejna-rewolucja/
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1736.pdf
[6] https://wandbox.org/permlink/yvna0fuNVilVmUAJ
[5] https://dev.krzaq.cc/post/check-out-the-new-issue-of-the-programista-magazine-polish/
[8] https://wg21.link/N4736
[7] https://wandbox.org/permlink/et8ct8fij3H1SOVP
[9] https://wandbox.org/permlink/fQFATr9ymfusm5yd
[A] https://github.com/ericniebler/range-v3
[B] https://programistamag.pl/programista-112018-78-styczenluty-2019-ethercat-deterministyczny-ethernet/
[C] https://fmt.dev/latest/index.html
[D] https://wandbox.org/permlink/q3m0OWiRZXJDe2AU
[E] https://wandbox.org/permlink/FCoXpUmUBqLRd17L
[F] https://wandbox.org/permlink/FIPoUlt653ZEpn6l
[10] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1143r2.html
[11] https://wandbox.org/permlink/p7eAM5iKYkogf7u0
[12] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1938r0.html
[13] https://wandbox.org/permlink/rnUyDVYvJHt9ZMZM
[14] https://wandbox.org/permlink/uOT8sSZRAwT8YTcy
[15] https://wandbox.org/permlink/PpADmpueggFGl2U6

PAWEŁ "KRZAQ" ZAKRZEWSKI

https://dev.krzaq.cc

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako Software Engineer w Backtrace I/O. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.