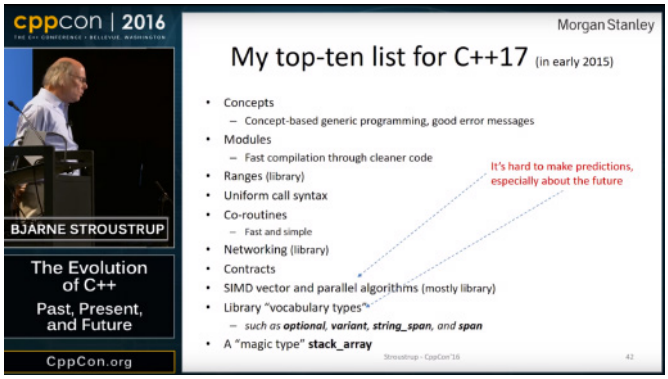
C++17 – nowy, miłościwie panujący nam standard C++

Nieco ponad rok temu (Programista 10/2016) zapowiedziany został nadchodzący standard języka. Od tego czasu komitet standaryzacyjny zdążył się jeszcze spotkać i dokonać pewnych zmian.

Finalny kształt C++17 poznaliśmy po marcowym spotkaniu komisji standaryzacyjnej w miejscowości Kona na Hawajach, gdy szkic standardu został poddany głosowaniu organów narodowych (ang. *national bodies*). 6 września głosowanie zostało zakończone jednogłośną akceptacją [1], co pozwoliło pominąć drugie głosowanie i przejść bezpośrednio do publikacji. Organizacja ISO opublikowała C++17 w grudniu 2017 jako ISO/IEC 14882:2017 [2]. Wedle relacji członków komitetu 9 miesięcy od zakończenia prac oznacza bardzo szybką publikację…

Od dawna wiadome było, że z szumnych zapowiedzi ewangelistów oraz z listy życzeń Bjarne Stroustrupa, oryginalnego twórcy języka, niewiele udało się zrealizować. Znakomicie obrazuje to *keynote* tego ostatniego na konferencji C++Con 2016, gdzie z wymienionych dziesięciu zmian żadna nie została w pełni zaimplementowana (Rysunek 1). Mimo to zmiany były liczne i w znaczącej większości pozytywne – choć w równie znaczącej większości drobne i nastawione na ułatwienie kodowania.

W tym artykule opisana zostanie ostateczna formuła nowego standardu. Zmiany uszeregowane będą mniej więcej od najistotniejszych dla programistów C++ do tych mniej ważnych, lub ważnych tylko dla specyficznych grup, np. twórców bibliotek.



Rysunek 1. Bjarne Stroustrup przedstawia listę życzeń dla C++17 (źródło: https://youtu.be/_wzc7a3McOs?t=3538)

USUNIĘCIE PRZESTARZAŁOŚCI

Wymienione tu części języka zostały *kompletnie usunięte* ze standardu, co oznacza, że kod je zawierający nie powinien się skompilować[1], ponieważ nie jest poprawnym kodem C++17 – tak samo jak kod zawierający przypisanie literału ciągu znaków do mutowalnego wskaźnika na `char` (Listing 1) nie powinien się skompilować w C++11 ani późniejszych [3].

1. Należy rozróżnić „nie powinien się skompilować" od „nie skompiluje się". Kompilatory nigdy w pełni restrykcyjnie nie trzymają się standardu, szczególnie przy domyślnych ustawieniach, więc czasem zamiast komunikatu błędu wyświetlą informację z ostrzeżeniem.

Listing 1. Niedozwolone przypisanie literału ciągu znaków do mutowalnego wskaźnika na char

```
int main()
{
  char* foo = "bar";
}
```

Specyfikacja throw()

Od teraz dozwolone jest wyłącznie `noexcept`. Wyjątkiem jest puste `throw()`, które staje się aliasem dla `noexcept(true)`. `throw(std::exception)` przedstawione w Listingu 2 jest niepoprawnym kodem.

Listing 2. Dynamiczna specyfikacja wyjątków

```
void foo() throw() {}
void bar() throw(std::exception) {}
```

Auto_ptr

`std::auto_ptr` to potworek pozostały po C++98, gdzie niemożliwa była poprawna implementacja `std::unique_ptr`. Wedle wiedzy autora we wszystkich sensownych zastosowaniach `std::auto_ptr` można zastąpić `std::unique_ptr`.

Listing 3. Użycie typu nieistniejącego już w bibliotece standardowej: std::auto_ptr<int>

```
int main()
{
  std::auto_ptr<int> a(new int);
}
```

Operator ++ dla bool

Samo istnienie operatora ++ dla `bool` może być dla wielu osób zaskoczeniem, choć od strony implementacyjnej wydaje się ono zrozumiałe, ponieważ `bool` jest w C++ (oraz w C) realizowany jako zmienna liczbowa o wielkości 1 bajta. W Listingu 4 przedstawiono niedozwolone od C++17 użycie.

Listing 4. Wykorzystanie operatora ++ dla bool

```
int main()
{
  bool b = false;
  b++;
  assert(b == true);
}
```

Trójznaki

Trójznaki (ang. *trigraphs*) to pozostałość po burzliwym rozwoju komputerów w latach 70-tych i 80-tych, znajdująca się w C++ dla kompatybilności z C, co było szczególnie istotne na początku istnienia języka. Komputery z tamtej epoki bardzo się od siebie wzajemnie różniły, a jedną z tych różnic były znaki dostępne na klawiaturach i zestawach znaków różnych platform.

Aby umożliwić korzystanie ze znaków `[]{}|~#^\`, które nie były dostępne na wszystkich platformach, C wprowadziło trójznaki (Tabela 1). Są to specjalne sekwencje trzech znaków, które były zamieniane na niedostępne na klawiaturze danego komputera znaki. Jedną z wymienianych wad, poza znacząco obniżoną czytelnością, jest zamiana trójznaków w pierwszej fazie translacji programu. Oznacza to, że zamieniane są przez kompilator przed czymkolwiek innym, nawet makrami preprocesora.

Obecnie nie są one prawie nigdzie wykorzystywane, nawet międzynarodowy konkurs zobfuskowanego (czyli takiego, którego czytanie zostało celowo utrudnione) kodu w C (ang. *The International Obfuscated C Code contest*) sugeruje ich unikanie [5]. Przykład ich złośliwego wykorzystania znajduje się w Listingu 5, gdzie `??/` zamieniane jest na \, co powoduje wciągnięcie wyrażenia warunkowego do komentarza i bezwarunkowe wywołanie funkcji poniżej.

Sekwencja	Zamieniana na
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Tabela 1. Trójznaki (źródło: [4])

Listing 5. Użycie trójznaku ??/ w celu wywołania trzeciej wojny światowej [6]

```
void launch_nuclear_missiles();

int main()
{
  bool we_are_at_war = false;
  // only send nuclear missiles if we're at war
  // we don't want needless deaths, do we??/
  if(we_are_at_war)
    launch_nuclear_missiles();
}
```

Poza trójznakami C++ ma jeszcze dwuznaki (ang. *digraphs*) oraz zamianę specjalnych tokenów. Więcej przeczytać można o tym w [25] [26].

INICJALIZACJA W WYRAŻENIU WARUNKOWYM

Jest to uproszczenie dla programistów, pozwalające na zapisanie wewnątrz instrukcji warunkowych `if` i `switch` inicjalizacji obiektu oraz warunku, co pozwala na uniknięcie dodatkowych zagnieżdżeń, jeśli obiekt używany jest tylko w części kodu wykonywanej warunkowo. Na przykład kod w C++ z Listingu 6 może zostać zastąpiony tym z Listingu 7. Analogicznie można inicjalizować obiekty w warunku `switch`.

Listing 6. Przykładowy kod w C++14

```
int main()
{
  map<int, int> graph;

  {
    auto result = graph.insert(make_pair(0, 42));
    if(result.second) {
      // stuff
    }
  }
}
```

Listing 7. Kod analogiczny do tego z Listingu 6, korzystający z nowości w C++17

```
int main()
{
  map<int, int> graph;

  if(auto result = graph.insert(make_pair(0, 42)); result.
  second) {
    // stuff
  }
}
```

Ciekawostką dla niektórych może być informacja, że już pierwszy standard C++, C++98, zezwalał na inicjalizację w warunkach wyrażeń warunkowych i pętli – ale tylko jeśli świeżo zdefiniowany obiekt był konwertowalny do wartości logicznej. W większości przypadków ograniczało to użyteczność do funkcji, które zwracały `nullptr` lub `0` w przypadku niepowodzenia, a takich, wbrew pozorom, nie ma wiele. Przykładowe użycie znajduje się w Listingu 8.

Listing 8. Inicjalizacja wewnątrz warunku, poprawna od początku ustandaryzowanego C++

```
void foo(void*);

int main()
{
  if(void* ptr = malloc(1048576)) {
    foo(ptr);
    free(ptr);
  }
}
```

STRUCTURED BINDINGS

Pierwotną formalną nazwą tej nowinki było *decomposition declarations*, choć potocznie wszyscy – wraz z twórcami – nazywali ją *structured binding declarations*. W marcu komitet standaryzacyjny ujednolicił nazewnictwo w tym zakresie, przyjmując popularniejszą potoczną nazwę.

Deklaracja *structured bindings* zezwala na przypisanie w jednej deklaracji zmiennych do elementów inicjalizatora, bez jawnego tworzenia dodatkowych zmiennych. „Rozpakowane" mogą zostać kontenery standardowe o statycznie znanej wielkości (`std::tuple`, `std::pair`, `std::array`) oraz typy zdefiniowane przez użytkownika, jeśli w takim typie:

» wszystkie niestatyczne elementy są dostępne publicznie, lub są elementami jego jednoznacznej publicznej klasy bazowej, oraz nie zawiera on anonimowych unii, lub

» jeśli oferuje poprawną specjalizację `std::tuple_size`, `std::tuple_element` i `get` (nie `std::get`, tylko `get` dostępne jako element klasy lub funkcja dostępna przez ADL[2]).

Deklaracja wygląda następująco: `auto [z1, z2, …, zn] = foo;`. Dodatkowo `auto` może być opatrzone referencją kwalifikowaną `const` lub `volatile`. Taką deklarację można porównać do utworzenia nienazwanej zmiennej, do której aplikowane są kwalifikatory przy `auto`, i użyciu jej do inicjalizacji kolejnych referencji do elementów nienazwanej zmiennej. Brzmi to trochę zawile, ale jest to łatwo zobrazować poprzez przykładowe zestawienie analogicznych fragmentów kodu ze standardów C++14 i C++17, które znajdują się w Listingach 9 i 10.

Listing 9. Kod w C++14

```
int main()
{
  auto tmp1 = std::make_pair("answer"s, 42);
  auto& a = tmp1.first;
  auto& b = tmp1.second;

  auto&& tmp2 = std::make_tuple("answer"s, 42, true);
  auto& c = std::get<0>(tmp2);
  auto& d = std::get<1>(tmp2);
  auto& e = std::get<2>(tmp2);

  auto const& tmp3 = std::make_tuple(1);
  auto& f = std::get<0>(tmp3);

  struct foo{ int bar; std::string baz; };
  foo qux{42, "answer"};
  auto& tmp4 = qux;
  auto& g = tmp4.bar;
  auto& h = tmp4.baz;

  int arr[2] = {42, 43};
  auto&& tmp5 = arr;
  auto& i = tmp5[0];
  auto& j = tmp5[1];
}
```

Listing 10. Kod analogiczny do tego z Listingu 9, z użyciem structured bindings

```
int main()
{
  auto [a, b] = std::make_pair("answer"s, 42);
  auto&& [c,d,e] = std::make_tuple("answer"s, 42, true);

  auto const& [f] = std::make_tuple(1);

  struct foo{ int bar; std::string baz; };
  foo qux{42, "answer"};
  auto& [g, h] = qux;

  int arr[2] = {42, 43};
  auto&& [i, j] = arr;

}
```

W Listingu 11 przedstawiona jest definicja własnej specjalizacji w celu zapewnienia obsługi *structured bindings* dla typu zdefiniowanego przez człowieka (ang. *user-defined*), którego liczba rozpakowanych elementów różni się od liczby elementów klasy.

Listing 11. Własne tuple_size/tuple_element/get

```
namespace kq
{

struct foo
{
  virtual ~foo() = default;

  virtual std::string const& name() const=0;

  virtual int id() const=0;
};

template<size_t> auto get(foo const&);

template<> auto get<0>(foo const& f)

{
  return f.id();
}

template<> auto get<1>(foo const& f)
{
  return f.name();
}

struct bar : foo
{
  bar(std::string const& n, int i): name_(n), id_(i) {}

  std::string const& name() const override
  {
    return name_;
  }

  int id() const override
  {
    return id_;
  }

private:
  std::string name_;
  int id_;
};

}

namespace std
{

template<>
class tuple_size<kq::foo>:
  public integral_constant<size_t, 2> {};

template<size_t I>
class tuple_element<I, kq::foo>
{
public:
  using type = decltype(get<I>(declval<kq::foo>()));
};

}

int main()
{
  kq::foo const& f = kq::bar{"answer", 42};

  auto&& [id, name] = f;

  std::cout << id << ", " << name << '\n';
}
```

Używając *structured bindings*, można poprawić czytelność kodu z Listingu 7. W Listingu 12 przedstawiono takie usprawnienie.

Listing 12. Przykład z Listingu 7 wzbogacony o structured bindings

```
int main()
{
  map<int, int> graph;

  if(auto [it, success] = graph.insert(make_pair(0, 42));
  success) {
    // stuff
  }
}
```

NOWE TYPY POMOCNICZE W BIBLIOTECE STANDARDOWEJ

Chodzi tutaj o `std::any`, `std::optional` oraz `std::variant`. Są one bardzo zbliżone do typów o analogicznych nazwach, ale różnią się nieznacznie semantyką i dostępnymi funkcjami.

2. Argument Dependant Lookup, znany również jako Koenig Lookup [A].

Any

`std::any` (z nagłówka <any>) to typ opakowujący, potrafiący przetrzymywać dowolny przenaszalny lub kopiowalny typ. Aby odzyskać wartość, należy użyć `std::any_cast` ze statycznie znanym typem. Jeśli podany typ będzie niepoprawny, zostanie rzucony wyjątek `std::bad_any_cast`. Przykładowe użycie znajduje się w Listingu 13.

Listing 13. Przykładowe użycie std::any [7]

```
int main()
{
  std::any foo;
  foo = 42;
  std::cout << std::any_cast<int>(foo) << std::endl;
  foo = "answer"s;
  std::cout << std::any_cast<std::string>(foo) << std::endl;
}
```

Optional

Zdefiniowany w nagłówku <optional> typ `std::optional` służy do opakowywania wartości, które mogą być potencjalnie puste, np. po wywołaniu API zakończonym niepowodzeniem. Jest to zbliżone podejście do zwracania `nullptr` jako wskaźnika, tylko w ustandaryzowany sposób.

W odróżnieniu od wersji z Boost nie posiada on funkcji `get()`, zastępując ją `value()` oraz `value_or()`. Do odzyskania obiektu można też użyć przeładowanego operatora dereferencji (`operator*()`), lub bezpośrednio uzyskać dostęp do elementów przechowywanego obiektu za pomocą syntaktyki wskaźnikowej, udostępnionej przez przeładowany `operator->()`. Sprawdzenie, czy obiekt przechowuje wartość, odbywa się przez `explicit operator bool()` lub funkcję `has_value()`.

Przykładowe użycie `std::optional` znajduje się w Listingu 14.

Listing 14. Przykładowe użycie std::optional [8]

```
int main()
{
  std::optional<std::pair<int, std::string>> foo =
    std::make_pair(0, "foo"s);
  if(foo) {
    std::cout << foo->first << ", "
      << (*foo).second << std::endl;
  }
}
```

Variant

Zwany też unią z tagiem (ang. *tagged union*) `std::variant` (<variant>) można rozumieć jako unię wiedzącą, który typ jest w niej w danym momencie zapisany. Pozwala to na uniknięcie powielania własnych implementacji tego podstawowego rozwiązania.

Sprawdzenie aktualnie przetrzymywanego typu odbywa się za pomocą funkcji `index()` lub `holds_alternative()`, a odzyskanie wartości za pomocą funkcji `std::get()` albo `get_if()`. Przykładowe użycie znajduje się w Listingu 15.

Listing 15. Przykładowe użycie std::variant [9]

```
int main()
{
  std::variant<int, std::string> foo = "foo"s;
  assert(std::holds_alternative<std::string>(foo));
  std::cout << std::get<std::string>(foo) << std::endl;
}
```

Opisane powyżej funkcje działają i są formalnie poprawne. Jednak idiomatycznie `std::variant` należy używać wraz z funktorami wizytującymi, za pomocą funkcji `std::visit()`. Dzięki temu można uniknąć drzewka if-ów, zastosować rozwiązania szablonowe, podzielić logicznie kod, stosując te same wizytatory dla różnych specjalizacji `std::variant`. Przykładowe użycie w Listingu 16.

Listing 16. Użycie std::visit [B]

```
struct visitor
{
  void operator()(int& val) const {
    std::cout << "int: " << val << '\n';
  }

  template<typename T> void operator()(T&& t) const {
    std::cout << "templated: " << t << '\n';
  }
};

int main()
{
  std::variant<int, double, std::string> foo = 42;

  std::visit(visitor{}, foo);

  foo = 4.76;
  std::visit(visitor{}, foo);

  foo = "foo"s;
  std::visit(visitor{}, foo);
}
```

STRING VIEW

Programiści C++ byli przyzwyczajeni do dość niewygodnego obchodzenia się z łańcuchami znaków, mając, bez dodatkowych bibliotek, dwa dostępne sposoby:

- C-stringi, które w większości zastosowań sprowadzają się do wskaźników na pierwszy element ciągu, czyli `char const*`. Ich największym problemem jest to, że długość napisu nie jest przekazywana wraz ze wskaźnikiem, więc często zbędnie powtarzane jest obliczanie tejże długości, a przekazanie wycinka napisu wymaga innego wywołania.
- Użycie typu `std::string`, ale jego wadą jest wymuszanie zbędnych kopii i alokacji (pomijając optymalizację małych stringów, ang. *small string optimization* [C] [D]).

C++17 wprowadza nową klasę – widok na ciąg znaków: `std::string_view`. Nie odpowiada ona za zwolnienie zasobów, a wyłącznie za przekazanie informacji o początku i długości ciągu. Wiele z funkcji operujących na `std::string` zostało wyposażonych w przeładowania akceptujące także `std::string_view`. Dodany również został literał `sv` do tworzenia instancji `std::string_view` z literałów ciągów znakowych. Przykład użycia znajduje się w Listingu 17.

Listing 17. Przykład użycia std::string_view [E]

```
void foo(std::string_view v)
{
  std::cout << v << "\n"sv;
}

int main()
{
  auto bar = "baz"s;
  foo(bar);
}
```

DEDUKCJA PARAMETRÓW SZABLONÓW KLAS

Dedukcja typów w szablonach funkcji była obecna w C++ od pierwszego standardu, ale aby utworzyć obiekt, należało jawnie podać parametry, co często prowadziło do duplikacji informacji zawartych w kodzie. Aby tego uniknąć, powstały obejścia w postaci serii funkcji `make_x`, tworzących instancję odpowiednio skonkretyzowanego typu szablonowego na podstawie przekazanych argumentów. Przykłady tego podejścia można znaleźć w Listingu 18, a w Listingu 19 przedstawiono jawne przekazanie parametrów szablonów.

Listing 18. Użycie funkcji std::make_pair i std::make_tuple [10]

```
int main()
{
  auto p = std::make_pair(42, "answer"s);
  auto t = std::make_tuple(p, "foo"s, 3.14);
}
```

Listing 19. Kod analogiczny do tego z Listingu 18, ale bez użycia funkcji make_x. [10]

```
int main()
{
  std::pair<int, std::string> p{42, "answer"s};
  std::tuple<
    std::pair<int, std::string>,
    std::string,
    double
  > t{p, "foo"s, 3.14};
}
```

C++17 pozwala pozbyć się większości funkcji z tej rodziny[3], wprowadzając dedukcję typów dla szablonów klas. Techniczna specyfikacja tego zachowania jest dość skomplikowana (więcej w: [F]), ale sprowadza się do tego, że kompilator porównuje argumenty przekazane jako inicjalizatory do dostępnych konstruktorów. Dzięki temu kod z Listingu 19 można znacząco uprościć, co przedstawiono w Listingu 20.

Listing 20. Uproszczenie kodu z Listingu 19 [11]

```
int main()
{
  std::pair p{42, "answer"s};
  std::tuple t{p, "foo"s, 3.14};
}
```

Jeśli powiązanie między inicjalizatorem a parametrem szablonu jest nietrywialne, na przykład w przypadku `std::vector` inicjalizowanego parą iteratorów, twórca klasy może zdefiniować własne sugestie w postaci prowadnic (ang. *deduction guides*). Przykład na podstawie klasy `foo` znajduje się w Listingu 21.

Listing 21. Definiowanie własnych deduction guides [12]

```
template<typename T, size_t I>
struct foo
{
  static constexpr auto value = I;
  using type = T;

  template<typename U> foo(T, U){}
};

template<typename T, typename U>
foo(T, U) -> foo<T, U::value>;

int main()
{
  foo f{
        std::string{},
        std::integral_constant<int, 42>{}
  };

  static_assert(f.value == 42);
  static_assert(
    std::is_same_v<decltype(f)::type, std::string>
  );
}
```

3. Należy jednak zauważyć, że nie wszystkie funkcje pasujące do tego nazewnictwa przestały być użyteczne – nie jest tak np. w przypadku `std::make_unique` lub `std::make_shared`. W uproszczeniu zasada brzmi następująco: jeśli funkcja `make_x` wymagała jawnego podania parametru szablonowego, to nadal może być przydatna.

ALGORYTMY WSPÓŁBIEŻNE

Wiele algorytmów dostępnych w nagłówkach `<algorithm>` i `<numeric>` dostało alternatywny zestaw przeładowań akceptujący klasy definiujące zasady wykonania (ang. *execution policy*). Standard definiuje trzy takie klasy, ale poszczególne implementacje mogą oczywiście wzbogacić wybór:

- `std::execution::sequenced_policy` – wykonanie sekwencyjne,
- `std::execution::parallel_policy` – wykonanie może być zrównoleglone,
- `std::execution::parallel_unsequenced_policy` – wykonanie może być zrównoleglone i zwektoryzowane.

Standard dostarcza globalne instancje powyższych klas, których należy używać, aby wybrać zasadę paralelizacji. Są to odpowiednio `std::execution::seq`, `std::execution::par` i `std::execution::par_unseq`.

We wszystkich wymienionych przypadkach kolejność nie jest sprecyzowana, co oznacza, że wywołanie korzystające z `std::execution::sequenced_policy` nie jest równoważne klasycznemu wywołaniu.

Przykładowe użycie znajduje się w Listingu 22. Autor zaznacza jednak, że w chwili publikacji artykułu żaden kompilator jeszcze nie wspierał tej części nowego standardu.

Listing 22. Przykładowe wykorzystanie zrównoleglonego wykonania

```
int main()
{
  std::vector foo{1,2,3,4,5};
  auto result = std::reduce(
    std::execution::par,
    foo.cbegin(),
    foo.cend(),
    std::multiplies{}
  );
  std::cout << result << '\n';
}
```

FILESYSTEM

Nowy nagłówek wzorowany na bibliotece o identycznej nazwie wchodzącej w skład Boosta. W końcu programiści C++ mogą za pomocą biblioteki standardowej operować na systemie plików. Przykładowy kod znajduje się w Listingu 23.

Listing 23. Przykładowe użycie biblioteki filesystem [13]

```
#include <filesystem>
#include <initializer_list>
#include <iostream>

namespace fs = std::filesystem;
```

```
int main()
{
  fs::create_directory("test");
  for(auto const& el : fs::directory_iterator(".")) {
    std::cout << el << '\n';
  }
}
```

Uwaga: w chwili pisania tego artykułu najnowsza wersja gcc ma tylko eksperymentalną implementację tej biblioteki, co powoduje, że jest ona faktycznie w nagłówku `<experimental/filesystem>`, przestrzeni nazw `std::experimental::filesystem` oraz wymaga linkowania biblioteki `stdc++fs` (`-lstdc++fs` do opcji linkera).

USPRAWNIENIA ATRYBUTÓW, NOWE ATRYBUTY

W nowym standardzie komitet kontynuuje sukcesywne rozszerzanie przydatności wprowadzonych w C++11 atrybutów:

- od teraz atrybuty mogą być nadawane przestrzeniom nazw i pojedynczym wartościom w enumeracjach (Listing 24),
- uściślono, co kompilator powinien robić, napotykając nieznane atrybuty – powinien je po prostu ignorować,
- wprowadzono alternatywny uproszczony zapis atrybutów w przestrzeniach nazw (Listing 25),
- wprowadzono trzy nowe standardowe atrybuty (wszystkie znajdują się w Tabeli 2):
  - `[[fallthrough]]`,
  - `[[maybe_unused]]`,
  - `[[nodiscard]]`.

Listing 24. Atrybuty w nowych miejscach

```
namespace [[deprecated("use foo")]] bar
{
}

enum class Foo
{
  foo,
  bar [[deprecated("foo > bar")]]
};
```

Listing 25. Alternatywny zapis atrybutów w porównaniu do klasycznego

```
[[long_namespace_name::foo, long_namespace_name::bar]]
void foo();

[[using long_namespace_name: foo, bar]]
void bar();
```

IF CONSTEXPR

Dotychczas w C++, tak jak i w wielu innych językach, kod w obu odnożach instrukcji warunkowej `if` musiał być poprawny zarówno syntaktycznie, jak i semantycznie. Było to wymagane, nawet gdy w trakcie kompilacji wartość wyrażenia warunkowego była znana i stała, powodując, że jedna z odnóg będzie martwym kodem.

Przykładowy niekompilujący się kod znajduje się w Listingu 26. Kompilator poinformuje, że `int` nie ma metody `size()` – niezależnie od tego, że nie ma możliwości, aby była ona na tym typie wywołana.

Tradycyjnym rozwiązaniem tego problemu było podzielenie kodu odpowiedzialnego za pracę na typach o różnych charakterystykach na różne szablony funkcji i *tag dispatching* pomiędzy nimi. Przedstawiono to w Listingu 27. Typ `std::is_same<T, std::string>` dziedziczy po `std::true_type` albo `std::false_type`, w zależności od tego, czy podane typy są identyczne. Ponieważ między tymi dwoma typami nie ma możliwości konwersji, inicjalizowane jest tylko jedno przeładowanie `foo_impl()`, poprawne dla danego T.

Atrybut	std	Opis
`[[carries_dependency]]`	C++11	Informacja dla optymalizatora o zachowaniu zależności `std::memory_order`
`[[deprecated]]` `[[deprecated("why")]]`	C++14	Oznaczenie przestarzałości, z opcjonalnym powodem
`[[fallthrough]]`	C++17	Informacja, że brak instrukcji `break;` pomiędzy `case`'ami w `switch`u jest celowy
`[[maybe_unused]]`	C++17	Informuje kompilator, że zmienna może pozostać nieużyta i nie jest to pomyłka programisty
`[[nodiscard]]`	C++17	Informacja, że rezultat tak oznaczonej funkcji nigdy nie powinien być ignorowany. W przypadku aplikacji do typu jest to aplikowane do wszystkich funkcji zwracających obiekty tego typu
`[[noreturn]]`	C++11	Określa, że wykonanie funkcji nigdy nie wróci do obecnego zakresu wykonania, np. `std::terminate()`

Tabela 2. Atrybuty w C++

Listing 26. Niepoprawny kod, int nie ma metody size(), nawet jeśli nigdy nie jest ona wywoływana [15]

```
template<typename T>
void foo(T t)
{
  if(std::is_same_v<T, std::string>) {
    std::cout << "string: " << t <<
      " size: " << t.size() << '\n';
  } else {
    std::cout << "other type: " << t << '\n';
  }
}

int main()
{
  foo("123"s);
  foo(456);
}
```

Listing 27. Poprawiony kod z Listingu 26 – kompilacja przebiega poprawnie i działa zgodnie z zamierzeniami [16]

```
template<typename T>
void foo_impl(T t, std::true_type)
{
  std::cout << "string: " << t <<
    " size: " << t.size() << '\n';
}

template<typename T>
void foo_impl(T t, std::false_type)
{
  std::cout << "other type: " << t << '\n';
}

template<typename T>
void foo(T t)
{
  foo_impl(t, std::is_same<T, std::string>{});
}

int main()
{
  foo("123"s);
  foo(456);
}
```

C++17 wprowadza `if constexpr`, który rezygnuje z wymogu semantycznej poprawności kodu odnóg – musi się on jedynie poprawnie parsować. Wobec tego bogate w słowa rozwiązanie z Listingu 27 można zamienić na to z Listingu 28. Różni się ono od tego z Listingu 26 dodaniem tylko jednego słowa: `constexpr`.

Listing 28. If constexpr [17]

```
template<typename T>
void foo(T t)
{
  if constexpr(std::is_same_v<T, std::string>) {
    std::cout << "string: " << t <<
      " size: " << t.size() << '\n';
  } else {
    std::cout << "other type: " << t << '\n';
  }
}

int main()
{
  foo("123"s);
  foo(456);
}
```

Jeśli warunków jest więcej, należy umieścić `constexpr` po każdym if-ie. Może to wydawać się zbyteczne, ale jest to spowodowane tym, jak gramatyka C++ określa `else if`. Może to być zaskakujące nawet dla doświadczonych programistów, ale takiego konstruktu nie ma wcale. Kod z Listingu 29 jest rozumiany przez kompilator tak samo jak ten z Listingu 30. Konieczność używania `constexpr` po każdym if-ie powinna być w tym momencie zrozumiała.

Listing 29. Drabinka if-ów

```
if(a)
  foo();
else if(b)
  bar();
else
  baz();
```

Listing 30. Kod z Listingu 29 w interpretacji kompilatora

```
if(a) {
  foo();
} else {
  if(b) {
    bar();
  } else {
    baz();
  }
}
```

FOLD EXPRESSIONS

Znana programistom języków funkcyjnych rodzina funkcji wyższego rzędu *fold* została wprowadzona do C++. Dzięki temu nie ma konieczności pisania pseudo-rekurencyjnych wywołań z coraz mniejszą liczbą argumentów funkcji. Dostępne są cztery formy wyrażeń *fold* (ang. *fold expression*):

- $(\ldots @ E)$ – jednoargumentowy *left fold* – $((E_1 @ E_2) @ \ldots) @ E_n$
- $(E @ \ldots)$ – jednoargumentowy *right fold* – $E_1 @ (\ldots @ (E_{n-1} @ E_n))$
- $(V @ \ldots @ E)$ – dwuargumentowy *left fold* – $(((V @ E_1) @ E_2) @ \ldots) @ E_n$
- $(E @ \ldots @ V)$ – dwuargumentowy *right fold* – $E_1 @ (\ldots @ (E_{n-1} @ (E_n @ V)))$

W Listingu 31 przedstawiono przykładową funkcję sumującą wszystkie argumenty bez zastosowania wyrażeń *fold*. Autor zwraca uwagę, że kod już został uproszczony dzięki `if constexpr`. W Listingu 32 przedstawiono analogiczny kod z ich wykorzystaniem.

Listing 31. Sumowanie parametrów funkcji bez wyrażeń fold

```
template<typename T, typename... Us>
auto sum_params(T t, Us... us)
{
  if constexpr(sizeof...(Us) == 0) {
    return t;
  } else {
    return t + sum_params(us...);
  }
}

int main()
{
  std::cout << sum_params(1, 2L, 3.f, 4.0, 5);
}
```

Listing 32. Wyrażenia fold w praktyce

```
template<typename... Us>
auto sum_params(Us... us)
{
  return (... + us);
}

int main()
{
  std::cout << sum_params(1, 2L, 3.f, 4.0, 5);
}
```

AUTO TEMPLATES

Jest to kolejna zmiana mająca na celu zwiększenie czytelności kodu. Dotychczas, aby użyć parametru szablonu niebędącego typem, należało podać jego typ. Jeśli typ ten nie był znany, musiał być innym parametrem szablonu. Za przykład może posłużyć tutaj obecny w bibliotece standardowej typ `std::integral_constant`, który musi być konkretyzowany poprzez podanie zarówno typu, jak i wartości, np. `std::integral_constant<size_t, 42>`. Od C++17 parametr szablonu może być oznaczony jako `auto`, wtedy jego typ będzie dedukowany z podanego argumentu.

Powyższy przypadek można uznać jeszcze za w miarę czytelny, ale już w przypadku wskaźnika na funkcję jest to znacząco mniej wygodne. W Listingu 33 przedstawiono życiową sytuację – opakowanie wskaźnika na funkcję celem przekazania go jako parametru *deleter* do `std::unique_ptr` [18]. W Listingu 34 przedstawiono semantycznie ekwiwalenty kod, w opinii autora znacznie czytelniejszy.

Listing 33. Definicja foo jako unique_ptr do przetrzymywania FILE*

```
template<typename T, T* func>
struct function_caller
{
  template<typename... Us>
  auto operator()(Us&&... us) const
    -> decltype(func(std::forward<Us...>(us...)))
  {
    return func(std::forward<Us...>(us...));
  }
};

using foo =
  std::unique_ptr<
    FILE,
    function_caller<decltype(fclose), &fclose>
  >;
```

Listing 34. Kod odpowiadający temu z Listingu 33, z parametrem szablonu auto

```
template<auto func>
struct function_caller
{
  template<typename... Us>
```

```
  auto operator()(Us&&... us) const
  {
    return func(std::forward<Us...>(us...));
  }
};

using foo = unique_ptr<FILE, function_caller<&fclose>>;
```

INNE ZMIANY

W tej sekcji znajdują się zmiany istotne dla konkretnych grup programistów (np. twórcy bibliotek) lub drobne, choć dostatecznie istotne, aby o nich wspomnieć.

Zmiana znaczenia listy inicjalizacyjnej z auto

Od teraz, przy inicjalizacji dedukowanego typu (`auto`) bez znaku równości, lista inicjalizacyjna może mieć tylko jeden element i to jego typ zostanie użyty. Choć oficjalnym powodem tej zmiany jest zwiększenie intuicyjności, w opinii autora jest dokładnie odwrotnie, ponieważ złamana została zasada, że inicjalizacja `typ nazwa{init}` jest równoznaczna z `typ nazwa = {init}`.

	C++14	C++17
auto foo{1};	std::initializer_list<int>	int
auto foo{1,2};	std::initializer_list<int>	Niepoprawny kod
auto foo = {1};	std::initializer_list<int>	std::initializer_list<int>
auto foo = {1,2};	std::initializer_list<int>	std::initializer_list<int>

Tabela 3. Dedukcja typów w C++14 i C++17

Static assert bez wiadomości

Jeśli warunek jest oczywisty, od teraz można wywoływać `static_assert` bez podawania wiadomości.

Ranged for z różnymi typami begin/end

Jest to zmiana mająca na celu ułatwienie wprowadzenia biblioteki ranges, lub implementację własnych bibliotek o podobnych funkcjonalnościach. Dzięki innemu typowi końca zakresu można w prosty sposób zmienić zachowanie operatora porównania i, na przykład, oznajmić, że łańcuch znakowy się kończy, gdy jego ostatni znak to null, bez względu na jego pozycję. Bardzo dokładnie opisał to Eric Niebler na swoim blogu [1A] [1B] [1C] [1D].

Noexcept wchodzi do sygnatury funkcji

Od C++17 `noexcept` jest częścią typu funkcji, przez co w pełni poprawny kod C++14 z Listingu 35 spowoduje błąd kompilacji w najnowszym standardzie. Jest tak, ponieważ funkcje `foo` i `bar` są typu `void()` w C++14, ale `bar` jest typu `void() noexcept` w C++17.

Listing 35. Zobrazowanie zmian w języku – noexcept staje się częścią typu funkcji

```
void foo() {}
void bar() noexcept {}

int main()
{
  using foo_t = decltype(foo);
  using bar_t = decltype(bar);
  static_assert(std::is_same<foo_t, bar_t>::value, "");
}
```

Akceptacja słowa kluczowego typename w szablonowych parametrach szablonów

W deklaracji szablonu, którego parametrem był kolejny szablon, tego zewnętrznego szablonu nie można było nazwać, używając słowa kluczowego `typename`, a wyłącznie `class`. Od C++17 już można.

Listing 36. Kod poprawny również w C++14

```
template<template <typename T> class foo>
struct bar{}
```

Listing 37. Kod poprawny tylko w C++17

```
template<template <typename T> typename foo>
struct bar{};
```

Zmienne inline

Od C++17 można stosować słowo kluczowe *inline* do definicji zmiennych globalnych. Zgodnie z nieoczywistym znaczeniem tego słowa kluczowego [1E] również w tym przypadku chodzi o zezwolenie na wielokrotną definicję zmiennej bez łamania zasady pojedynczej definicji (ang. *One-definition rule*).

W Listingu 38 przedstawiono zastosowanie zmiennych *inline* w C++17. W Listingu 39 można znaleźć emulację zbliżonego rozwiązania za pomocą szablonów w C++11.

Listing 38. Zmienne inline

```
inline std::string const version = "1.0.0.42";

struct foo
{
  inline static std::string mut = "foo";
};
```

Listing 39. Rozwiązanie poprawne w C++11 zbliżone do tego z Listingu 38

```
template<typename>
struct foo_impl
{
  static std::string mut;
};

template<typename T>
std::string foo_impl<T>::mut = "";

using foo = foo_impl<void>;
```

Nowe funkcje matematyczne

Często używana do obliczania odległości na kartezjańskiej siatce współrzędnych funkcja `std::hypot()` doczekała się przeładowania dla trzech parametrów. Od C++17 pojawiła się w bibliotece standardowej funkcja do obliczania najmniejszej wspólnej wielokrotności – `std::lcm()`, oraz największego wspólnego dzielnika – `std::gcd()`. Warta uwagi jest również funkcja `std::clamp()`, zamykająca wartość w podanym przedziale – można ją zobrazować w następujący sposób:

Listing 40. Pseudokod obrazujący działanie funkcji std::clamp

```
template<typename T>
auto clamp(T a, T b, T c)
{
  return std::max(b, std::min(a, c));
}
```

to_chars/from_chars

Dostępne w nagłówku `<charconv>` funkcje o mocno ograniczonej funkcjonalności przeznaczone do, odpowiednio, serializacji i deserializacji liczb. Dzięki temu, że nie rzucają wyjątków, nie obsługują *locale* oraz nie alokują pamięci, można spodziewać się ich wysokiej wydajności. Czyni je to kandydatami do użycia w bibliotekach serializujących, np. json, i w systemach wbudowanych, gdzie wysoka wydajność jest ważniejsza niż lekko kuriozalne API.

W Listingu 41 pokazano przykładowe użycie tych funkcji, wraz z nietypowym sposobem, w jaki można sprawdzać, czy wywołanie się powiodło.

Listing 41. Wykorzystanie from_chars i to_chars [28]

```
int main()
{
  char buf[] = "42.5";
  int val;
  auto r = std::from_chars(buf, buf+sizeof(buf), val);

  if(auto&& [ptr, err] = r; !bool(err)) {
    // prints 42
    std::cout << val << '\n';

    char out[32] = {};
    // base 21
    auto r = std::to_chars(out, out+32, val, 21);
    if(auto&& [ptr, err] = r; !bool(err)) {
      auto distance = ptr - out;
      // prints 20 (42 is 20_21)
      std::cout << std::string_view{out, distance};
    }
  }
}
```

Gwarantowana optymalizacja RVO

RVO – *return value optimization* (również *copy elision*) [1F] – to optymalizacja, która była opcjonalna od C++98, a teraz staje się obowiązkowa. Jeśli inicjalizowany jest obiekt wynikiem funkcji, która inicjalizuje obiekt tego samego typu (lub konwertowalnego do niego) w instrukcji `return`, kompilator miał prawo (teraz obowiązek) nie wygenerować obiektów tymczasowych, tylko zainicjalizować obiekt docelowy, pomijając przy tym wykonanie konstruktorów kopiujących/przenoszenia. Przed C++17 te konstruktory musiały być dostępne, ponieważ ta optymalizacja była opcjonalna; od teraz nie są, co pozwala zwracać nieprzenaszalne i niekopiowalne klasy z funkcji. W Listingu 42 pokazano to na przykładzie `std::mutex`.

Listing 42. Zwracanie nieprzenaszalnego i niekopiowalnego typu std::mutex z funkcji

```
std::mutex make_mutex()
{
  return std::mutex{};
}

int main()
{
  auto m = make_mutex();
}
```

Using z ...

Celem ułatwienia np. dziedziczenia po operatorach wywołania wszystkich argumentów paczki szablonu, `using` może korzystać z rozpakowywania za pomocą operatora trzech kropek.

Listing 43. Using z ...

```
template<typename... T>
struct simple_overloader : T...
{
  simple_overloader(T... ts) : T{ts}... {}
  using T::operator()...;
};

int main()
{
  simple_overloader foo{
    [](int){ return 42; },
    [](double x){ return x * 2; }
  };

  std::cout << foo(0) << ", " <<
    foo(668.5) << '\n';
}
```

Tablicowy shared_ptr

Wprowadzono obecne w `std::unique_ptr` przeładowanie dla tablic.

Listing 44. Tablicowy shared_ptr

```
int main()
{
  std::shared_ptr<std::string[]> x(new std::string[10]);

  x[0] = "foo";
}
```

Constexpr lambdy

Od C++17 lambdy mogą znajdować się w wyrażeniach obliczanych w trakcie kompilacji. Są też niejawnie deklarowane jako `constexpr` przez kompilator, jeśli istnieje taka możliwość.

Kopia *this w lambdach

Wprowadzono możliwość przekazania `*this` jako elementu listy obiektów przekazanych do wyrażenia lambda. Dzięki temu wewnątrz lambdy nie trzeba się martwić czasem życia obiektu zewnętrznego.

Listing 45. Przekazanie *this do lambdy

```
struct foo
{
  int x;

  auto cpp14()
  {
    return [copy = *this]{
      return copy.x;
    };
  }

  auto cpp17()
  {
    return [*this]{
      return this->x;
    };
  }
};

int main()
{
  auto a = foo{42}.cpp14();
  auto b = foo{42}.cpp17();

  std::cout << a() << ", " <<
    b() << '\n';
}
```

Przenoszenie elementów map/setów bez realokacji

Klasy `std::set`, `std::map`, `std::multiset`, `std::multimap` oraz ich odpowiedniki z *unordered* w nazwie wzbogacone zostały o funkcje `merge()` i `extract()`, a ich funkcje `insert()` o przeładowania obsługujące typ zwracany przez `extract()`. Pozwalają one na przenoszenie elementów pomiędzy różnymi obiektami tych klas, gdy spełnione zostaną następujące warunki:

- kontenery są tego samego typu lub są swoimi odpowiednikami zezwalającymi lub zabraniającymi powtarzania kluczy,
- typ elementu jest identyczny,
- typ alokatora jest identyczny.

Porównanie lub hash nie muszą się zgadzać, więc można przenosić elementy np. pomiędzy odwrotnie posortowanymi instancjami `std::set`, co pokazano w Listingu 46.

Listing 46. Użycie extract() i insert()

```
int main()
{
  std::set<int, std::greater<>> a{1,2,3};
  std::set<int, std::less<>> b{2,3};

  b.insert(a.extract(1));

  assert((b == std::set<int, std::less<>>{1,2,3}));
  assert((a == std::set<int, std::greater<>>{2,3}));
}
```

Krótszy zapis zagnieżdżonych przestrzeni nazw

Od C++17 zapis `namespace foo::bar {}` jest poprawny i nie trzeba zapisywać każdej przestrzeni ręcznie: `namespace foo{ namespace bar {} }`.

__has_include

Wprowadzony został standardowy sposób na sprawdzenie, czy dany plik nagłówkowy jest dostępny w momencie kompilacji. Pozwala to na użycie biblioteki tylko jeśli jest dostępna, lub dostarczenia zastępczej funkcjonalności, jeśli preferowanej brak. Sztuczny przykład znajduje się w Listingu 47.

Listing 47. Użycie __has_include

```
#if __has_include(<memory>)
#include <memory>
template<typename T>
using shared_ptr = std::shared_ptr<T>;
#elif __has_include(<boost/shared_ptr.hpp>)
#include <boost/shared_ptr.hpp>
template<typename T>
using shared_ptr = boost::shared_ptr<T>;
#else
static_assert(false, "no shared ptr available");
#endif
```

Algorytmy szukające

Funkcja `std::search()` zyskała przeładowanie akceptujące własne algorytmy szukające. Wraz z tym w standardzie zdefiniowano trzy takie algorytmy:

- `std::default_searcher` – domyślny algorytm szukający z biblioteki standardowej,
- `std::boyer_moore_searcher` – algorytm szukający Boyera-Moore'a dla stringów,
- `std::boyer_moore_horspool_searcher` – algorytm szukający Boyera-Moore'a-Horspoola dla stringów.

Listing 48. Przykład użycia std::boyer_moore_searcher [20]

```
int main()
{
  auto txt = "foo bar baz"s;
  auto sought = "bar"s;

  std::boyer_moore_searcher searcher{
    sought.begin(),
    sought.end()
  };

  auto result = std::search(
    txt.begin(),
    txt.end(),
    searcher
  );

  if(result != txt.end()) {
    auto pos = std::distance(txt.begin(), result);
    std::cout << "Found at position " << pos << '\n';
  } else {
    std::cout << "Not found\n";
  }

}
```

Ułatwienia w metaprogramowaniu

Wprowadzone zostają standardowe implementacje funkcji `std::apply()`, `std::invoke()` oraz typu `std::void_t`. Obok wprowadzonych do standardu C++14 typów `std::index_sequence`/`std::integer_sequence` są one podstawą metaprogramowania. Chociaż wszystkie wymienione można z powodzeniem zaimplementować już w C++11, to standaryzacja zezwoli na swobodne ich wykorzystanie w kodzie, bez obaw o wielokrotną implementację – ani bez jej konieczności.

`std::invoke()` oferuje stały interfejs do wywoływania funkcji, zbliżony do tego z `std::bind()` lub `std::thread`. Pierwszym parametrem jest *callable*, czyli dowolny wywoływalny obiekt lub wskaźnik. Jeśli pierwszym argumentem jest wskaźnik do niestatycznej funkcji klasy, to następnym argumentem jest obiekt, na którym zostanie on wywołany. Pozostałe argumenty to argumenty przekazywane do wywoływanej funkcji. Pozwala to na ujednolicenie wywołań przy pisaniu generycznych funkcji wyższego rzędu.

Listing 49. Przykładowe użycie std::invoke [21]

```
int main()
{
  auto ptr = &std::string::size;
  std::cout << std::invoke(ptr, "123"s) << '\n';

  char up = std::invoke(&::toupper, 'x');
  std::cout << up << '\n';
}
```

`std::apply()` pozwala na przekazanie jako argumentów funkcji elementów klasy `std::tuple` lub innej, zgodnej z nią. Zgodna klasa posiada odpowiednią specjalizację `std::tuple_size` oraz wspiera funkcję `std::get()`.

Listing 50. Przykładowe użycie std::apply [22]

```
int main()
{
  std::tuple t{2, 10.0};
  std::cout << std::apply(&::pow, t);
}
```

`std::void_t` to alias dowolnego typu lub sekwencji typów na `void`. Jest to bardzo użyteczne narzędzie w powiązaniu ze SFINAE[4] i pozwala na odrzucanie niepoprawnego kodu bez powodowania błędu kompilacji. W Listingu 51 zawarto przykład oparty na przykładzie z cppreference [23].

Listing 51. Przykładowe użycie std::void_t [24]

```
template<class, class = std::void_t<>>
struct has_type_member:
   std::false_type{};

template<class T>
struct has_type_member<T, std::void_t<typename T::type>>:
   std::true_type{};

int main()
{
  std::cout << has_type_member<int>{} << '\n' <<
    has_type_member<std::false_type>{};
}
```

Polimorficzne alokatory

C++17 wprowadził polimorficzne alokatory w przestrzeni nazw `std::pmr`. Wraz z nimi w tej przestrzeni pojawiły się odpowiednio dostosowane kontenery, np. `std::pmr::vector`. Polimorficzne alokatory, jak nazwa wskazuje, mogą wykazywać różne zachowanie dla różnych instancji tego samego typu alokatora. Pozwala to na lepszą współpracę własnych alokatorów z kontenerami standardowymi.

Zarezerwowane przestrzenie nazw

Zarezerwowano dla biblioteki standardowej C++ wszystkie przestrzenie nazw, które można opisać wyrażeniem regularnym `/::std\d*/`. Czyli na przykład `std`, `std2` lub `std2017` w globalnej przestrzeni nazw.

std::launder

Nazwa tej funkcji jest analogią do prania pieniędzy (ang. *money laundering*), lecz odnosi się do pamięci. Tak jak wyprane pieniądze stają się znów legalnym środkiem płatniczym, ponieważ służby nie są w stanie wyśledzić ich pochodzenia, tak „wyprana" pamięć może być użyta do innych celów, ponieważ kompilator „zapomina", co w niej wcześniej było. Ma to istotne znaczenie, gdy w danym miejscu znajdował się obiekt, którego odczyty kompilator miał prawo zoptymalizować.

Jej zastosowanie pokazano w Listingu 52 zaczerpniętym z serwisu StackOverflow [27]. Bez zastosowania `std::launder()` asercja mogłaby zakończyć się niepowodzeniem, ponieważ kompilator miał prawo zoptymalizować dostępy do `u.x.n`, które jest stałą.

Listing 52. Użycie std::launder()

```
int main()
{
  struct X { const int n; };
  union U { X x; float f; };

  U u = {{ 1 }};

  X *p = new (&u.x) X {2};

  assert(*std::launder(&u.x.n) == 2);
}
```

4. Niemożność podstawienia [parametru szablonu] nie jest błędem [kompilacji] (ang. *Substitution Failure Is Not An Error*).

PODSUMOWANIE

Pomimo że niniejszy artykuł jest obszerny, to i tak nie opisano w nim wszystkich zmian i nowości w C++17. W opinii autora te najważniejsze zostały jednak poruszone. Ocenę, czy C++17 spełnia pokładane w nim nadzieje – zarówno pod kątem obiecanych funkcjonalności, jak i jako z założenia znaczący standard – autor pozostawia czytelnikom.

W sieci:

[1]: https://herbsutter.com/2017/09/06/c17-is-formally-approved/
[2]: https://www.iso.org/standard/68564.html
[3]: https://goo.gl/78oE9C
[4]: https://goo.gl/TbTv7Y
[5]: https://www.ioccc.org/2013/guidelines.txt
[6]: https://wandbox.org/permlink/Oolg8K4wZxRPZtlP
[7]: https://wandbox.org/permlink/RPSCKS5GpbWiLS79
[8]: https://wandbox.org/permlink/pRflCz6TTMn78yBZ
[9]: https://wandbox.org/permlink/AFBAirwnaKmDBjx8
[A]: http://en.cppreference.com/w/cpp/language/adl
[B]: https://wandbox.org/permlink/HNlExe4tTV9nwij4
[C]: https://stackoverflow.com/a/21710033/2456565
[D]: https://youtu.be/kPR8h4-qZdk
[E]: https://wandbox.org/permlink/AC8e1O3OUzfsN1Kt
[F]: https://goo.gl/vw4vo8
[10]: https://wandbox.org/permlink/Hia6pmnsgDHv0kMk
[11]: https://wandbox.org/permlink/fOVWvngwyEBhbMg9
[12]: https://wandbox.org/permlink/EWeanhXbjz6qCoPm
[13]: https://wandbox.org/permlink/a9xsieNnobtWhh8Q
[14]: https://wandbox.org/permlink/ZYznuRoTkdIVv5FA
[15]: https://wandbox.org/permlink/FGsJRqEXc9gZcmmp
[16]: https://wandbox.org/permlink/bKQtrDhrcvANG6I9
[17]: https://wandbox.org/permlink/SyBUipzYxmBy3Izl
[18]: https://goo.gl/k1XD4A
[19]: https://timsong-cpp.github.io/cppwp/n4659/diff.cpp14.dcl.dcl
[1A]: http://ericniebler.com/2014/02/16/delimited-ranges/
[1B]: http://ericniebler.com/2014/02/18/infinite-ranges/
[1C]: http://ericniebler.com/2014/02/21/introducing-iterables/
[1D]: http://ericniebler.com/2014/02/27/ranges-infinity-and-beyond/
[1E]: https://dsp.krzaq.cc/post/352/co-oznacza-slowo-kluczowe-inline/
[1F]: https://en.wikipedia.org/wiki/Copy_elision
[20]: https://wandbox.org/permlink/6OMuiOs3deo4ilPi
[21]: https://wandbox.org/permlink/uRcg3ufDpz5gsn9z
[22]: https://wandbox.org/permlink/p5e8tzlxkbS6bKKw
[23]: http://en.cppreference.com/w/cpp/types/void_t
[24]: https://wandbox.org/permlink/kAsnMoyGIX5gSVHl
[25]: http://en.cppreference.com/w/cpp/language/operator_alternative
[26]: https://goo.gl/BwvCWv
[27]: https://stackoverflow.com/a/39382728/2456565
[28]: https://wandbox.org/permlink/2ZQzVQq2oXANcOIz

PAWEŁ "KRZAQ" ZAKRZEWSKI
https://dev.krzaq.cc

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako starszy programista C++ w firmie Huuuge Games. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.