# C++17

## Co znajdziemy w nowym standardzie

Jest już niemal pewne, że następnym standardem języka C++ będzie C++17. Czy będzie on – zgodnie z oczekiwaniami – równie wielkim trzęsieniem ziemi jak C++11? Niestety, zgodnie z prawem nagłówków Betteridge'a odpowiedź na to pytanie jest przecząca. Ale i tak jest z czego się cieszyć.

### WSTĘP

Ostatnie spotkanie komitetu standaryzacyjnego w roku 2016 za nami (Issaquah, listopad 2016), dzięki czemu można już z dużą pewnością przewidywać ostateczny kształt standardu C++17 – wraz z jego nazwą. W artykule opisane zostały co istotniejsze (w mniemaniu autora) zmiany w języku, jego bibliotece standardowej oraz zmiany, których się spodziewano, ale których nie udało się wprowadzić w wyznaczonym czasie.

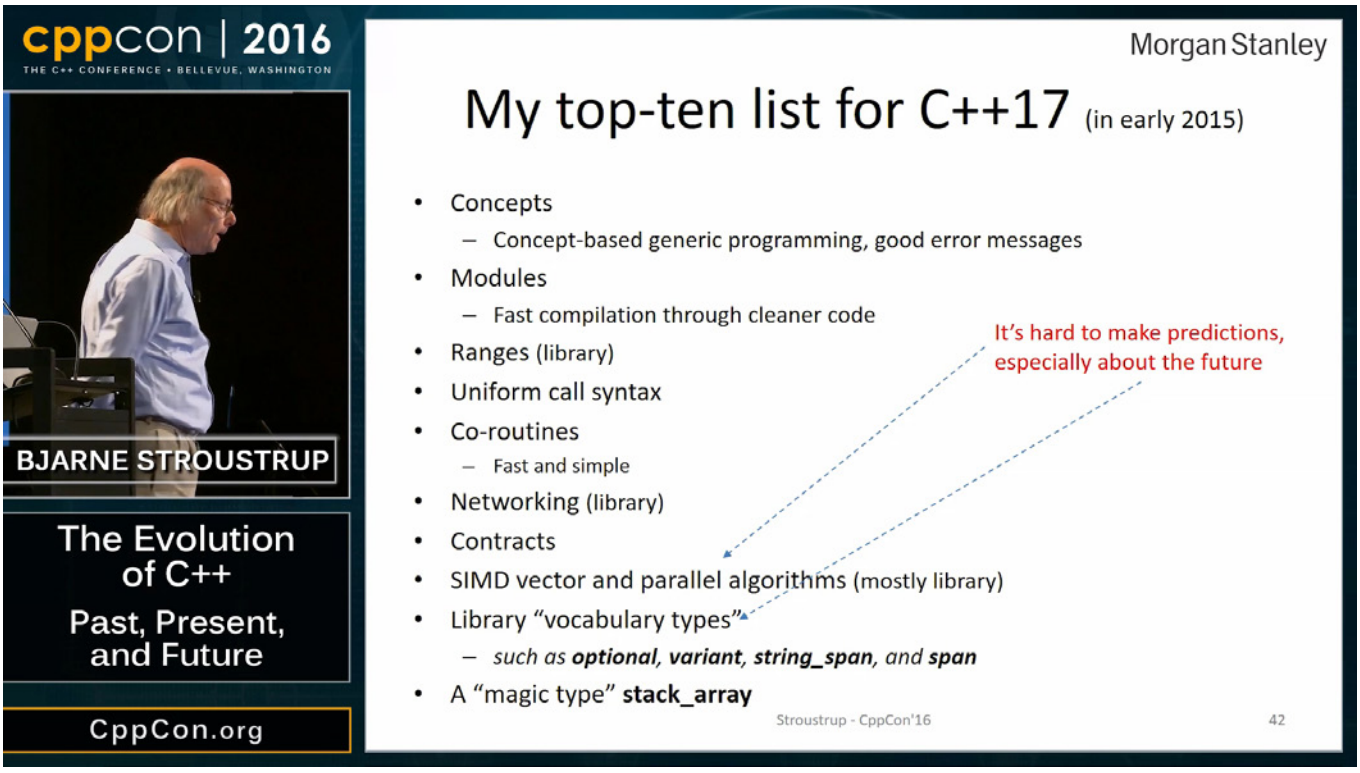
### KANDYDACI DO C++20

W swojej prelekcji na konferencji CppCon we wrześniu 2016 roku. Bjarne Stroustrup – oryginalny twórca języka – przedstawił swoją „listę życzeń" dla C++17 z roku 2015. Jak sam zauważył [1], z dziesięciu punktów udało się wdrożyć tylko dwa, w niepełnym wymiarze. Pozostałe są rozwijane jako specyfikacje techniczne i jest prawdopodobne, że będą częścią następnego standardu – C++20.

#### *Koncepty*

Jednym z popularnych paradygmatów programowania w C++ jest programowanie generyczne, które jest głównie oparte o zastosowanie szablonów i *duck typingu*. Dla przykładu, szablon funkcji `std::sort` przyjmuje jako iteratory parametry dowolnego typu, tak długo jak spełniają one wymagania opisane w standardzie. Jednakże jedyną weryfikacją ich zdatności jest faktyczne ich użycie. Może to być chociażby dereferencja lub inkrementacja. Pozwala to uniknąć wymagania dziedziczenia iteratorów po wyimaginowanej klasie `iterator` (dzięki temu wskaźnik też jest iteratorem) lub `less_than_comparable` przez elementy sortowanej kolekcji. Kosztem tego rozwiązania są długie i nieczytelne komunikaty błędów, zależne od tego, jak głęboko w stosie wywołań funkcji dojdzie do użycia obiektu w sposób, którego on nie obsługuje – nawet dla jednego problematycznego przypadku potrafią one być dłuższe niż nie tylko ten artykuł, ale i cały magazyn.

Wśród głównych założeń konceptów niezmiennie od debiutu tego pomysłu w 2003 r. znajduje się umożliwienie szybszego spraw-




*Rysunek 1. Bjarne Stroustrup, cele dla C++17 (źródło: https://youtu.be/_wzc7a3McOs?t=3543)*

dzania zdatności typów przekazywanych szablonom. W efekcie przekazywane użytkownikowi komunikaty błędów zyskują na czytelności.

*Concepts TS* [1] jest zaimplementowany w kompilatorze g++ w wersji 6.1 i nowszych. Dla przykładu, w Listingu 1 przedstawiono kod z wykorzystaniem konceptów zawierający błędne wywołanie funkcji `sort`. Komunikat błędu kompilatora pokazany jest w Listingu 2. Przy bezpośrednim wywołaniu funkcji `std::sort` ten sam kod z tymi samymi ustawieniami kompilatora generuje 9 kB tekstu w 85 liniach (ok. 15x więcej niż wersja z konceptami).

**Listing 1. Kod z wykorzystaniem konceptów**

```
#include <algorithm>
template <typename T>
concept bool
RandomAccessIterator = requires(T t) {
  *t;
  t++;
  t+0;
};
void sort(RandomAccessIterator b,
          RandomAccessIterator e) {
  std::sort(b, e);
}
int main() {
  sort(1,3);
}
```

**Listing 2. Komunikat błędu z konceptami**

```
> g++ m.cpp -std=c++1z -fconcepts 2>&1
m.cpp: In function 'int main()':
m.cpp:16:10: error: cannot call function
'void sort(auto:1, auto:1) [with auto:1 = int]'
  sort(1,3);
          ^
m.cpp:10:6: note:   constraints not satisfied
  void sort(RandomAccessIterator b, RandomAccessIterator e) {
       ^~~~
m.cpp:4:14: note: within ‘template<class T>
concept const bool RandomAccessIterator<T> [with T = int]'
  concept bool RandomAccessIterator = requires(T t) {
               ^~~~~~~~~~~~~~~~~~~~
m.cpp:4:14: note:     with 'int t'
m.cpp:4:14: note: the required expression '* t' would be
ill-formed
```

## Moduły

Obecne w C++ rozwiązanie problemu dołączania zewnętrznego kodu odziedziczone jest z języka C i nie zmieniło się od kilkudziesięciu lat. Polega ono na kopiowaniu przez preprocesor tekstowej zawartości plików nagłówkowych za pomocą dyrektywy `#include`. Od wielu lat znane są jego wady, z których jednymi z najczęściej wymienianych są:

- skalowalność czasów kompilacji – każdy użyty plik nagłówkowy musi być co najmniej raz obsłużony dla każdej jednostki translacji (ang. *translation unit)*, która go pośrednio lub bezpośrednio używa,
- wrażliwość – wykonywana jest prosta podmiana tekstowa, możliwe są kolizje nazw makr z nazwami identyfikatorów w kodzie użytkownika,
- brak logicznego powiązania z modułami w projekcie lub bibliotece,
- utrudnienie rozumienia kodu przez narzędzia (np. IDE) – pliki nagłówkowe bibliotek mają wiele sekcji wewnątrz dyrektyw `#ifdef`, które wyłączają je z kompilacji w zależności od ustawień systemowych lub preferencji użytkownika. Doprowadziło to do sytuacji, gdzie IDE muszą używać prawdziwego kompilatora, przekazując mu parametry kompilacji, aby uzyskać poprawny obraz kodu.

*Modules TS* [2] jest odpowiedzią na te problemy oraz wiele innych, oferując moduły podobne do tych znanych z języków C#, D lub Java. Przykładowy kod przedstawiający wykorzystanie modułów znajduje się w Listingu 3. Choć istnieją dwie implementacje modułów (w kompilatorach VC++ i clang), nie są one ze sobą w pełni zgodne. Podstawowe kwestie sporne między nimi to eksportowanie makr oraz wydajne budowanie drzewa zależności.

Dodatkowo, w czasie pisania tego artykułu, żaden z kompilatorów nie był w stanie skompilować programu wyświetlającego *Hello, World!*, wykorzystującego wyłącznie moduły.

**Listing 3. Przykładowy kod wykorzystujący moduły (źródło: N4610)**

```
import std.io;
module M;
export module std.random;
export struct Point {
  int x;
  int y;
};
```

## Biblioteka zakresów (ranges)

Jednym z podstawowych konceptów w bibliotece standardowej C++ jest iterator. Jest to schemat typu pozwalający na poruszanie się po abstrakcyjnych kolekcjach danych, najczęściej wskazujący na konkretne miejsce w kolekcji lub zaraz poza nią. Pomimo że jest to bardzo potężna i wszechstronna abstrakcja, użycie iteratorów jako podstawowego budulca biblioteki standardowej niesie za sobą szereg problemów:

- większość algorytmów (np. `std::find`, `std::count`) operuje na kolekcjach, więc trzeba im podać iterator początku i końca,
- z powyższego powodu niemożliwe jest przekazanie wyniku jednego algorytmu bezpośrednio do kolejnego bez używania zmiennych tymczasowych,
- powoduje duplikację kodu,
- stosunkowo niebezpieczne – przekazanie iteratorów do różnych kontenerów może łatwo pozostać niezauważone i powodować problemy po uruchomieniu, wykazując niezdefiniowane zachowanie,
- utrudniona optymalizacja – zachowywanie wyników pośrednich bardziej skomplikowanych algorytmów do zmiennych tymczasowych utrudnia lub uniemożliwia proces optymalizacji.

Odpowiedzią na te problemy jest *Ranges TS* [3]. Biblioteka range-v3 przygotowywana przez Erica Nieblera jest dostępna w serwisie GitHub [4] i kompatybilna z większością współczesnych kompilatorów. Przykładowy kod korzystający z biblioteki zakresów znajduje się w Listingu 4.

**Listing 4. Przykładowy kod korzystający z range-v3**

```
#include <range/v3/all.hpp>
int main() {
  std::vector<int> v{10,9,8,7,6,5,4,3,2,1,0};
  using namespace ranges;
  copy(action::sort(v) | view::remove_if([](int n) {
    return n % 2;
  }), ostream_iterator<int>(std::cout, ", "));
  // wypisze "0, 2, 4, 6, 8, 10, "
}
```