AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

C++ 17

Czyli nowoczesny C++ na przykładach

Jarosław Cierpich
Arkadiusz Kasprzak

Plan prezentacji

- Część I: Arkadiusz Kasprzak – m.in. dedukcja typów w szablonach klas, fold expressions
- Część II: Jarosław Cierpich – m.in. structured binding i constexpr if

Część I

Fold expressions

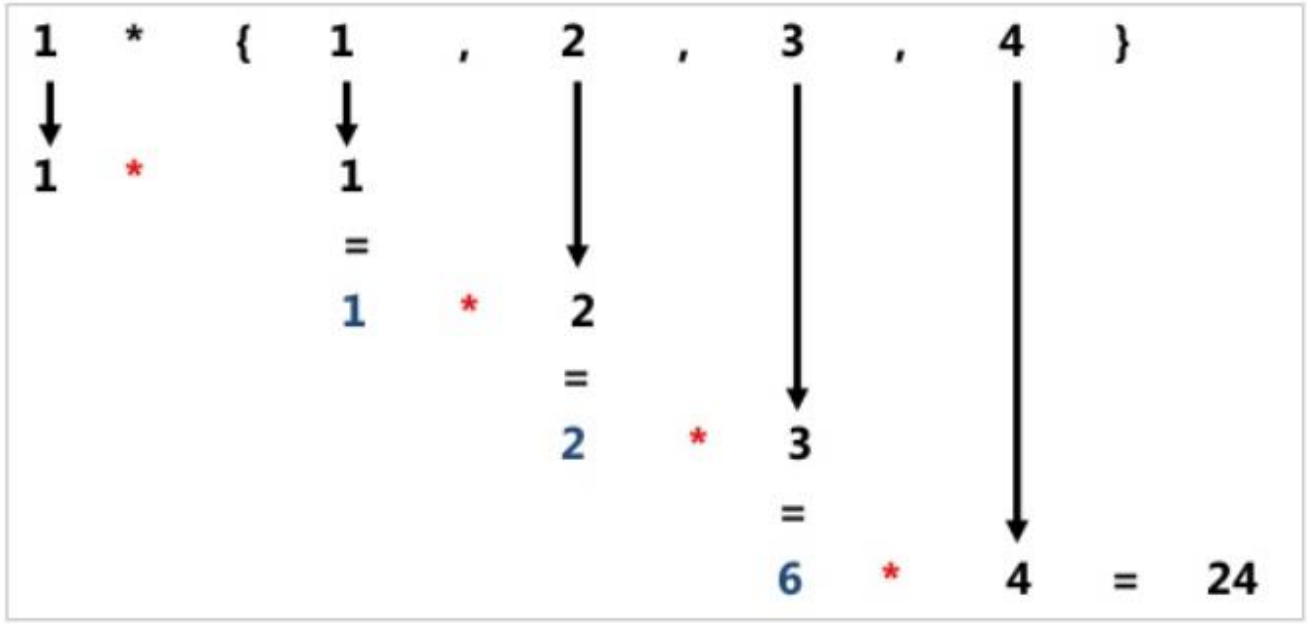
Fold expressions

- Krok w stronę funkcyjnego C++
- Fold (ang. zwijać, składać) – rodzina funkcji wyższego rzędu występująca w językach funkcyjnych, które działają na kolekcjach (traktując je w sposób rekurencyjny). Rekurencyjnie przetwarzają kolekcję, „redukując” ją i stopniowo tworząc wynik za pomocą dwuargumentowej funkcji łączącej (np. operator).

Źródło: https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Fold expressions

Źródło: https://www.slideshare.net/sermp/functional-programminginc



Fold expressions

- W C++17 pojawia się funkcjonalność związana z parameter packiem, którą nazywamy fold expressions
- Polega ona na przetworzeniu elementów parameter packa za pomocą jakiegoś operatora dwuargumentowego.
- Rozwinięcie koncepcji związanych z variadic templates – uproszczenie składni przy jednoczesnym dodaniu nowych możliwości.

- przykład: fold_1.cpp

Fold expressions

- Zaleta: krótszy kod

4 rodzaje zapisu:

(pack op …)

(… op pack)

(pack op … op init)

(init op … op pack)

Wydaje się skompilkowane, ale jest proste tylko trzeba zrozumieć co się dzieje.

Fold expressions

4 rodzaje zapisu:

(pack op …)

(… op pack)

(pack op … op init)

(init op … op pack)

pack – nierozwinięty parameter pack

op – operator dwuargumentowy (32 możliwe)

init – wartość początkowa

Jak myślicie, jaka jest różnica?

Fold expressions

- Różnica jest w nawiasowaniu, jakie zostanie zastosowane po rozwinięciu wyrażenia.
- Każdy z podanych wariantów ma swoją nazwę.

Fold expressions

- Unary right fold (pack op …)
- Rozwijane w sposób następujący:
- (arg_1 op (arg_2 op (… (arg_n1 op arg_n)))
- Czyli np. mając parameter pack: 1, 2, 3, 4 i operator +:

(1 + (2 + (3 + 4)))

Czyli najpierw używamy elementów „z końca” parameter packa

Fold expressions

- Unary left fold (… op pack)
- Rozwijane w sposób następujący:
- (((arg_1 op arg_2) op arg_3) … op arg_n)
- Czyli np. mając parameter pack: 1, 2, 3, 4 i operator +:

(((1 + 2) + 3) + 4)

Czyli najpierw używamy elementów „z początku” parameter packa

Fold expressions

- Pozostałe dwie wersje to tzw. binary (right/left) fold. Dokładamy w nich po prostu wartość, która zostanie użyta w pierwszym wykonaniu operacji (patrz slajd 9).
- Right: (arg_1 op (arg_2 … (arg_n op init)))
- Left: (((init op arg_1) op arg_2) … op arg_n)

- przykład: fold_2.cpp

Fold expressions

- Możemy mieć sytuację, że stosując unary fold, parameter pack będzie pusty
- Wtedy tylko kilka operatorów zapewnia poprawną wartość: &&, || oraz ,.
- Te wartości to kolejno: true, false i void().
- W każdym innym wypadku niepoprawna konstrukcja.

- przykład: fold_3-6.cpp

Dedukcja argumentów w szablonach klas (CTAD)

Class template argument deduction (CTAD)

- C++17 wprowadza dedukcję argumentów w szablonach klas – nie trzeba już podawać typów w sposób bezpośredni.
- Jedna z największych zmian w C++17
- Będziemy mówić o dedukcji argumentów, które są typami.
- Z założenia bardzo prosta, jak się okaże, niesie za sobą sporo zmian – w tym upraszcza znacznie pisany kod
- Najpierw przypomnimy sobie, jak to było wcześniej – żeby zrozumieć dlaczego ta zmiana była potrzebna.

Class template argument deduction (CTAD) – definicja problemu

- Przed C++17: konieczność specyfikowania typów przy tworzeniu obiektów za pomocą szablonów klas:
- std::pair<int, double> para (1, 3.5);
- Taki kod (gdzie mamy szablony klas, których argumenty są typami) pojawia się bardzo często w C++ - np. za każdym razem, gdy tworzyliśmy obiekt z szablonu std::vector lub innych kontenerów STL.
- Tutaj widać, że typy są łatwo dedukowalne – tak naprawdę podanie ich pomiędzy <> stanowi nadmiarową informację i tylko kompiluje kod.

Class template argument deduction (CTAD) – definicja problemu

- Tutaj było jasne, że chcemy int i double.
- Oczywiście nie każda sytuacja jest tak prosta, będziemy omawiać takie, gdzie typy nie są tak oczywiste.

Trochę powtórki – dedukcja typów przed C++17

- Dedukcja typów w szablonach jest już od dawna – już od standardu C++98 następuje ona dla szablonów funkcji.
- Można było pisać:

```
template<typename T>
T& min (const T& a, const T& b) {
      return a < b ? a : b;
}
int smaller = min (6,7);
```

- Nie musieliśmy podawać sami typów – kompilator potrafił wydedukować je sam, na podstawie podanych argumentów.

  - przykład: deduction_before.cpp

Trochę powtórki 2 – dedukcja typów przed C++17

- Podejście zaprezentowane na przykładzie ma jedną zasadniczą zaletę – trzeba pisać mniej kodu, który zawiera powtórzone informacje.
- Przed C++17 często musieliśmy więc podawać zbędne informacje.
- Oczywiście przed C++17 istniało rozwiązanie tego problemu – były to tzw. funkcje typu „make” - np. make_pair.
- Rozwiązanie to bazuje na dedukcji dla szablonów funkcji – pozwala użytkownikowi funkcji tworzyć obiekty klas bez podawania typów. Stanowi to jednak jedynie obejście, a nie pełne rozwiązanie problemu.

- przykład: make_function_idiom.cpp

Trochę powtórki 3 – dedukcja typów przed C++17

- Biblioteka Standardowa zawiera kilka tego typu funkcji – chyba najbardziej znane to std::make_pair i std::make_tuple.
- Problem jest taki, że nie każdy typ czy kontener takie funkcje posiada – nie ma ich np. std::vector. Funkcję tego typu można sobie oczywiście zaimplementować, ale nie zawsze jest to proste zadanie.

Rozwiązanie problemu - C++17

- Od standardu C++17 wspomniane problemy znikają – nie ma już konieczności podawania typów bezpośrednio czy pisania funkcji typu make_XYZ (choć te drugie przydają się jeszcze w rzadkich przypadkach).
- Pojawia się mechanizm Class Template Argument Deduction (CTAD) – czyli dedukcji typów podanych jako argumenty szablonu klasy na podstawie argumentów podanych do konstruktora tej klasy.

- przykład: deduction_now.cpp

CTAD

- Od C++17 możemy więc pisać tak:

```
std::pair result{"Student E"s, 5.0}; ///
   dedukcja do std::pair<std::string, double>
```

Mechanizm ten jest prosty w użyciu dla osoby korzystającej, jak tutaj, z gotowego rozwiązania – chcielibyśmy jednak wiedzieć, jak to działa.

CTAD

- Zasada: kompilator wydedukuje argumenty szablonu na podstawie typu w inicjalizacji:

- przy deklaracji z inicjalizacją zmiennej, np. std::pair p(3, 4.5);
- przy wyrażeniu z new: auto a = new std::pair {2, 3.5};
- przy rzutowaniu w stylu funkcyjnym (function – style cast).

CTAD – własne szablony klas

- Wszystkie dotychczasowe przykłady bazowały na gotowych szablonach z Biblioteki Standardowej – przyjrzyjmy się teraz własnej klasie:

- przykład: deduction_simple_own_class.cpp

CTAD – działanie

- W pokazanym przykładzie mechanizm zadziałał automatycznie
- Nie zawsze tak będzie -o czym potem
- Musimy zrozumieć, jak cały mechanizm działa pod spodem – jak kompilator radzi sobie z przeprowadzeniem dedukcji.
- Ważne: całość opiera się na znanych już zasadach: dedukcji argumentów dla szablonów funkcji.
- Jeśli tworzymy obiekt za pomocą szablonu klasy, nie podając typów, kompilator buduje sobie swego rodzaju „szkice” szablonów funkcji dla konstruktorów.

CTAD – działanie

- Za chwilę pokażę na przykładzie, jak to wygląda. Podam również algorytm.
- Rozróżniamy dwa podstawowe przypadki: gdy dedukcja zachodzi sama i programista nie musi robić nic, i gdy trzeba dopisać pewien kod, by dedukcja zadziałała.
- Ten pierwszy, dużo prostszy przypadek zachodzi, gdy konstruktor wykorzystuje w liście argumentów wszystkie parametry szablonu klasy w taki sposób, że można łatwo wydedukować typ (tzn. nie jest on np. zagnieżdżony w innym typie).

CTAD – algorytm

- Rozważamy sytuację, gdy mamy szablon klasy C. Tworzymy obiekt nie podając argumentów szablonowych. Dedukcja zachodzi w sposób następujący:
- Dla każdego konstruktora Ci kompilator tworzy FIKCYJNY szablon funkcji Fi (ten „szkic” -tzw. deduction candidates) w taki sposób, że:
  - Szablonowe argumenty Fi są szablonowymi argumentami C (czyli szablonu klasy), po których następują szablonowe argumenty konstruktora Ci, a następnie argumenty domyślne.
  - Parametry funkcji Fi (te pomiędzy ()) są parametrami takimi, jak w konstruktorze.
  - Typ zwracany to C.

Na koniec dodawany jest jeszcze fikcyjny szablon funkcji na podstawie konstruktora C(C) – copy deduction candidate.

CTAD – algorytm

- Dalej cały mechanizm działa już jak dla szablonów funkcji – bo dokładnie w taki sposób widzi sytuację kompilator.
- Zachodzi więc dedukcja dla szablonów klas oraz mechanizm overload resolution – dokładnie tak samo jak było to wcześniej.
- Teraz na przykładzie pokażę, jak wygląda cały proces.

CTAD – przykład

```
template <typename T1, typename T2, typename T3, typename T4>
class Storage{
public:
    Storage(const T1& first, const T2& second, const T3& third, const T4& fourth) :
      m_first(first), m_second(second), m_third(third), m_fourth(fourth) {
        std::cout << "L-value constructor" << std::endl;
    }

    Storage(T1&& first, T2&& second, T3&& third, T4&& fourth) : m_first(first),
      m_second(second), m_third(third), m_fourth(fourth)  {
        std::cout << "R-value constructor" << std::endl;
    }

private:
    T1 m_first; T2 m_second; T3 m_third; T4 m_fourth;
};

/// Uzycie:
Storage s1 {"Student A"s, 273322 , "IS"s, 5.0};
```

CTAD – przykład

```
Storage (const T1& first, const T2& second, const T3& third, const T4&
    fourth) : m_first(first), m_second(second), m_third(third),
    m_fourth(fourth)
{
   std::cout << "L-value constructor" << std::endl;
}

Storage (T1&& first, T2&& second, T3&& third, T4&& fourth) :
    m_first(first), m_second(second), m_third(third), m_fourth(fourth)
{
   std::cout << "R-value constructor" << std::endl;
}
```

Patrzymy na konstruktory jak na zwykłe funkcje.

CTAD – przykład

```
template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F1 (const T1& first, const T2& second, const T3& third, const T4& fourth) : m_first(first),
                    m_second(second), m_third(third), m_fourth(fourth)
{
   std::cout << "L-value constructor" << std::endl;
}

template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F2 (T1&& first, T2&& second, T3&& third, T4&& fourth) : m_first(first), m_second(second),
m_third(third), m_fourth(fourth)
{
   std::cout << "R-value constructor" << std::endl;
}

template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F3 (Storage<T1, T2, T3, T4>); // copy deduction candidate
```

CTAD – przykład

```
template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F1 (const T1& first, const T2& second, const T3& third, const T4& fourth) : m_first(first),
                    m_second(second), m_third(third), m_fourth(fourth)
{
   std::cout << "L-value constructor" << std::endl;
}

template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F2 (T1&& first, T2&& second, T3&& third, T4&& fourth) : m_first(first), m_second(second),
m_third(third), m_fourth(fourth)
{
   std::cout << "R-value constructor" << std::endl;
}

template <typename T1, typename T2, typename T3, typename T4>
Storage<T1, T2, T3, T4>
F3 (Storage<T1, T2, T3, T4>); // copy deduction candidate
```

CTAD – przykład

Był to przypadek najprostszy. Można sobie jednak wyobrazić dużo bardziej skompilowane – np. gdy konstruktor sam w sobie jest szablonem.

Nie omawiam z uwagi na czas, bo są jeszcze inne ważne rzeczy do omówienia.

Po więcej odsyłam do:

https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

Tam znaleźć można jeszcze co najmniej jeden interesujący przykład.

CTAD – all or nothing

- Pomiędzy <> podajemy wszystkie typy albo żadnego
- Wyjątek – argumenty domyślne (ich podawać nie trzeba wszystkich)
- Co ciekawe zasada ta nie działa dla szablonów funkcji

- przykłady:

deduction_all_or_nothing_1-3.cpp

No dobrze, ale...

- Do tej pory wszystko działało z naszego punktu widzenia samo.
- A co z tym przypadkiem:

```
std::vector a {1,2,3,4,5,6};
std::vector b (a.begin(), a.end());
```

Dedukcja działa dla kontenerów.

Czy to powyżej zadziała? Jeśli tak to dlaczego? Co tu jest dziwnego?

Przykład: deduction_vector_1.cpp

CTAD

- W przykładzie z poprzedniego slajdu konstruktor miał następującą sygnaturę:

```
template <typename Iterator>
vector (Iterator begin, Iterator end);
```

Więc widać, że podajemy tylko iterator – a jednak jakoś udało się dokonać dedukcji.

Oznacza to, że osoba pisząca vector dostarczyła jeszcze jakiejś dodatkowej informacji.

User defined deduction guides

- Podajemy zasady, na jakich ma dokonać się dedukcja dla konstruktorów o określonej sygnaturze.
- Bywa tak, że my wiemy, jak ma coś zostać wydedukowane – nie wie natomiast kompilator.
- Zasada: Jeśli konstruktor nie posiada argumentów które specyfikują wszystkie niedomyślne typy z argumentów szablonu, to trzeba kompilatorowi pomóc.

User defined deduction guides

- Składnia:

template_name (parameter_declaration) -> template_id;

- Czyli np.:

template <typename Iter> vector (Iter begin, Iter end)
->
vector<typename std::iterator_traits<Iter>::value_type>;

-przykład: user_defined_deduction_guides.cpp

User defined deduction guides

- Muszą być definiowane w tym samym zakresie co szablon klasy, do którego się odnoszą
- Muszą znajdować się po definicji szablonu klasy – kolejność ma znaczenie
- Nie muszą być szablonami. Przykład:

```
template <typename T>
class A { A(const T& elem){} }

A(const char*) -> A<std::string>;
```

Deduction guides dla kontenerów STL

- Uwaga na pułapkę:

```
std::vector b {a.begin(), a.end()};
```

- Inicjalizacja za pomocą listy: wydedukuje

```
std::vector<std::vector<int>::iterator>
```

- przykład: deduction_vector_2.cpp

- Ten sam sposób działania dla reszty kontenerów sekwencyjnych (poza array – z uwagi na przekazywanie rozmiaru trochę bardziej skomplikowany – bram tam również konstruktora jak powyżej).
- https://en.cppreference.com/w/cpp/container/vector/deduction_guides
- Zawiera spis deduction guides dla kontenerów.

Podsumowanie

- Bardzo skomplikowany i rozbudowany temat. Przedstawiony przeze mnie materiał to tylko wstęp.
- Po więcej odsyłam do cppreference.com
- … oraz do ciekawej prezentacji:

https://www.youtube.com/watch?v=STJExxBU54M

Backup

Słowo typename w szablonowych parametrach szablonów

„typename” w szablonowych param. szablonów

- Prosta zmiana.
- Szablonowe parametry szablonów – jedno z ciekawszych zastosowań szablonów w C++.
- Przydatne, gdy mamy zależne od siebie parametry w szablonie – np. tak jest w przypadku, gdy chcemy stworzyć adapter – pierwszy parametr reprezentuje przechowywany typ, a drugi – użyty kontener
- template <typename T, template <typename Elem, typename Allocator> class Cont> class fifo {};
- Dotychczas problem z miejscem zaznaczonym na czerwono.

„typename” w szablonowych param. szablonów

- Sytuacja przez standardem C++17: wymagane jest używanie słowa kluczowego class przy szablonowych parametrach szablonów – słowo kluczowe typename pomiędzy <> a nazwą typu powinno powodować błąd kompilacji
- gcc pozwala na kompilację nawet z flagą -std=c++14, ale już dodanie flagi -pedantic-errors daje błąd

- Przykład template_template_before.cpp

„typename” w szablonowych param. szablonów

- Jak widać jest to dość sztuczne ograniczenie, które jedynie komplikuje język - do tego stopnia, że było ignorowane przez kompilatory przed standardem C++17
- Zmiana w C++17: Od C++17 użycie słowa kluczowego typename jest w pełni zgodne ze standardem.

- Przykład template_template_now.cpp

Opcjonalna wiadomość w static_assert

Zmiana w static_assert

- static_assert wykonuje asercję w czasie kompilacji.
- Składnia przed C++17:
- static_assert(bool_constexpr, message);
- Od standardu C++17 wprowadzono drugą możliwość:
- static_assert(bool_constexpr);
- Znikła więc konieczność podawania wiadomości, która wyświetlała się jako błąd kompilacji w momencie, gdy wyrażenie w asercji zwróciło false.

- przykład: static_assert_change.cpp

Część II

Inline variables

Inline variables

Standard c++17 pozwala na zdefiniowanie zmiennych jako **inline**

Pozwala to na tworzenie zmiennych globalnych bez obawy o to, że zostały one wcześniej zdefiniowane w jakimś pliku, który dołączamy

Pozwala również w prosty sposób **definiować** statycznych członków klasy

Właściwości inline variables

Zmienna inline może mieć więcej niż jedną definicję tak długo, jak każda z nich występuje w osobnej jednostce translacyjnej.

Definicja zmiennej typu inline musi występować w każde jednostce translacyjnej, w której jest używana

Jeżeli występuje więcej niż jedna definicja to **w każdej jednostce translacyjnej** jest używana ta, która została napotkana jako pierwsza.

Przykład inline_variables.cpp

constexpr if

constexpr if

Pozwala na zdefiniowanie wyrażenia warunkowego, które zostanie wyewaluowane w czasie kompilacji

Warunek w takim wyrażeniu musi być konwertowalny do stałego wyrażenia typu bool

Kod jest generowany tylko dla warunku, który jest ewaluowany to **true**

Przykład constexpr_if_basic.cpp

Własności constexpr-if

constexpr nie jest zamiennikiem dla dyrektywy preprocesora #if

Przykład constexpr_if_disc_err.cpp

Constexpr if pozwala nam w prosty sposób zastąpić specjalizację

Przykład constexpr_if_templ.cpp

Ref qualifiers (c++11)

Ref qualifiers

Wprowadzone w standardzie 11. Są używane przy nie-statycznych metodach składowych klasy.

Pozwalają określić która metoda powinna zostać użyta w przypadku, gdy niejawny parametr metody, jakim jest obiekt na rzecz którego jest wywoływana metoda, jest r-wartością lub l-wartością.

Przykład ref_qualifiers.cpp

Structured bindings

Structured bindings declaration

- Jest to deklaracja, która pozwala związać (ang. bind) nazwy z „podobiektami” (ang. subobjects) lub elementami inicjalizującymi.

- Deklaracji tej możemy używać aby w prosty sposób „rozpakować” bardziej złożoną strukturę

- Sposób zapisu:
  (const volatile) auto(&|&&) [id1(, id2, id3…)] = wyrażenie (1)
  (const volatile) auto(&|&&) [id1(, id2, id3…)] (wyrażenie) (2)
  (const volatile) auto(&|&&) [id1(, id2, id3…)] {wyrażenie} (3)

- Przykład struct-bind-basic.cpp

Sposób określania typów

Tworzona jest „ukryta zmienna” **e**, której typ jest określany w jeden z dwóch sposobów:

Jeżeli wyrażenie jest typu tablicowego (**A**), a przy słowie kluczowym auto nie stoi operator referencji wtedy typ **e** jest określany w następujący sposób: **cv A**, gdzie cv to słowa kluczowe const i/lub volatile stojące przy słowie kluczowym auto.

W przeciwnym wypadku typ **e** jest taki, jak gdybyśmy mu przypisali wyrażenie (np.: auto& e = wyrażenie)

Sposób wiązania (ang. bind)

Na podstawie **e** jest określany typ **E**, gdzie
E = std::remove_reference_t(decltype(e));

WAŻNE! Wiązanie, podobnie jak referencja, jest aliasem do istniejącego obiektu, natomiast w przeciwieństwie do referencji nie musi być typem referencyjnym

Przykład struct-bind-not-ref.cpp

Następnie na podstawie E wyróżniamy trzy przypadki:

Jeżeli **E** jest typem tablicowym, to nazwy stojące między „[”, a „]” są bezpośrednio wiązane z elementami tablicy **e**

Przykład struct-bind-table.cpp

Sposób wiązania (ang. bind)

Jeżeli **E** jest klasą, która nie jest unią oraz std::tuple_size<E> jest określony, wtedy wykorzystywany jest mechanizm wiązania „tuple-like”

Ten sposób wiązania możemy wykorzystać, aby „uzbroić” naszą klasę we wsparcia dla structured binding

Przykład struct-bind-own-class.cpp

Sposób wiązania (ang. bind)

Jeżeli **E** jest klasą, która nie jest unią oraz std::tuple_size<E> jest nieokreślony, wtedy nazwy stojące między „[”, a „]” są wiązane z dostępnymi polami klasy

Przykład struct-bind-non-tuple.cpp

If – init-statement

If – init-statement

Standard C++ 17 pozwala nam na inicjalizację zmiennej w następujący sposób:

if(init; condition){…}

Zmienna zainicjalizowana w ten sposób może zostać wykorzystana w warunku

Zmienna jest widoczna **tylko** w zakresie tego wyrażenia (wychodzi poza zakres wraz z zamykającym „}”)

Przykład if_init_basic.cpp

Odpowiednik

```
if(init; cond){}
else {}
```

Jest odpowiednikiem

```
{
    init;
    if(cond){}
    else{}
}
```

If – init-statement

Przydatne użycia if-init statement

if_init_raii.cpp

if_init_structured_binding.cpp

Przykład użycia z switch

switch_init_basic.cpp

Dziękujemy za uwagę

Źródła

- Jacek Galowicz „C++17 STL Cookbok”
- Dimitri Nesteruk "Design Patterns in Modern C++"
- Strona cppreference.com : https://en.cppreference.com/w/cpp/language/fold
- http://eel.is/c++draft/temp.variadic#9
- https://riptutorial.com/cplusplus/example/14773/folding-over-a-comma
- https://www.reddit.com/r/cpp/comments/5kiqeb/understanding_fold_expressions/
- https://stackoverflow.com/questions/38060436/what-are-the-new-features-in-c17
- https://en.cppreference.com/w/cpp/language/class_template_argument_deduction
- https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/template_template_arguments.htm
- https://baptiste-wicht.com/posts/2015/05/cpp17-fold-expressions.html
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0091r3.html
- https://arne-mertz.de/2017/06/class-template-argument-deduction/#User-defined_deduction_guides
- https://en.wikipedia.org/wiki/C%2B%2B17
- https://www.bfilipek.com/2017/06/cpp17-details-templates.html#template-argument-deduction-for-class-templates
- https://dsp.krzaq.cc/post/1417/artykul-cpp17-nowy-milosciwie-panujacy-nam-standard-c-z-programisty-66/
- https://stackoverflow.com/questions/38060436/what-are-the-new-features-in-c17

Źródła część II

- Jacek Galowicz „C++17 STL Cookbok”
- https://en.cppreference.com/w/
- https://skebanga.github.io
- https://stackoverflow.com/
- https://www.codingame.com/