C++ 17

Czyli nowoczesny C++ na przykładach
Część II

Jarosław Cierpich
Arkadiusz Kasprzak

Plan prezentacji

- Część I: Arkadiusz Kasprzak – nowe typy danych: std::byte, std::string_view, std::any, std::variant, std::optional

- Część II: Jarosław Cierpich – invoke, nowe algorytmy, usprawnienia w std::map

Część I

Nowe typy danych w C++17

- W C++17 pojawiło się kilka nowych typów danych:
- std::byte
- std::string_view
- std::any
- std::variant
- std::optional

- Kilka z nich znane jest wielu programistom z zewnętrznych bibliotek (projekt Boost).

std::byte

std::byte

- Nowy typ wprowadzony w C++17
- Znajduje się w znanym wcześniej nagłówku <cstddef> (w tym nagłówku jest też m.in.. nullptr_t).
- Implementuje bajt zgodnie ze standardem C++ jako kolekcję bitów, bez żadnych dodatkowych interpretacji.

std::byte

- Wcześniej bajt reprezentowany jako char, unsigned char czy uint8_t.
- Typy te narzucają pewną interpretację przechowywanych bitów (np. dla char jest to jakiś znak).
- Nie zawsze chcemy, by taka interpretacja miała miejsce – nie zawsze ma sens wykonywanie operacji charakterystycznych np. dla char.
- std::byte ma zdefiniowane tylko operatory bitowe.

std::byte

- Sprawia to, że niepoprawna interpretacja przechowywanych w nim danych jest mniej prawdopodobna (np. uint8_t miało możliwość dodawania – co nie zawsze ma sens w przypadku bajtu – np. gdy ma on reprezentować jakiś rejestr przechowujący pewne informacje).
- std::byte reprezentuje więc czyste dane, co ma w teorii zmniejszyć liczbę błędów z powodu przypadkowej błędnej interpretacji.
- Przykład: byte_1.cpp

std::byte

- std::byte wspiera operacje bitowe: << i >>, <<= i >>=, |=, &= i ^= oraz | (suma), & (iloczyn), ^ (alternatywa wykluczająca) i ~ (negacja).
- Może zostać zainicjalizowany za pomocą wartości całkowitej, może również do niej zostać jawnie przekonwertowany za pomocą std::to_integer<T>();
- Przykład: byte_2.cpp - proste, ale troche mniej oczywiste zastosowanie.

std::byte

- Podsumowanie:
- std::byte to kolekcja nieinterpretowanych bitów.
- Przydatne, gdy chcemy uniemożliwić wykonywanie na danych operacji innych niż bitowe.
- Oczywiście powstaje pytanie: czy opłaca się wprowadzać taką dość kosmetyczną zmianę?

std::byte

- Wiele bibliotek używa dziś do reprezentacji bajta typów char lub uint8_t – i one raczej przy tych typach zostaną.
- std::byte jest natomiast bardzo dobrą opcją, gdy zaczynamy tworzyć nowy projekt opierający się w pewnym stopniu na bardziej niskopoziomowych operacjach – pozwala w czytelniejszy sposób wyrazić intencję programisty.

std::string_view

std::basic_string_view

- Szablon klasy reprezentującej obiekt który odwołuje się do niemodyfikowalnego łańcucha znakowego.
- non-owning – jest to tylko widok
- Widok na sekwencję znaków (string zarówno w stylu C jak i C++)
- std::basic_string_view jest to szablon bazowy, mamy 4 specjalizacje dla różnych typów znaków.

std::basic_string_view

- std::string_view, czyli std::basic_string_view<char> - o tym będziemy dziś mówić najwięcej
- std::wstring_view, czyli std::basic_string_view<wchar_t>
- std::u16string_view, czyli std::basic_string_view<char16_t>
- std::u32string_view, czyli std::basic_string_view<char32_t>
- std::u8string_view, czyli std::basic_string_view<char8_t> (C++20)

std::string_view

- Jaki jest powód wprowadzenia tego nowego typu?
- Wprowadzony został w celu optymalizacji – pozwala ograniczyć ilość przeprowadzanych kopiowań.
- Poza tym jest to bardzo „lekki” obiekt, ideowo wygląda tak:

```
class string_view {
  const char* m_str;
  size_t m_len;
};
```

- Mamy więc tylko dwie informacje.

std::string_view

- 4 konstruktory:

1) Domyślny constexpr basic_string_view() noexcept;

2) Kopiujący: constexpr basic_string_view(const basic_string_view& copied) noexcept = default; (może, bo tylko wskaznik i liczba – szybkość kopiowania)

3) constexpr basic_string_view (const CharT* s, size_type count) – widok pierwszych x znaków

4) constexpr basic_string_view(const CharT* s) – łańcuch z ‘\0’ na końcu

- Przykład: string_view_1.cpp

std::string_view

- Bardzo ważne – std::string_view i wszystkie jego operacje działają na stałych sekwencjach znaków – nie ma możliwości dokonania modyfikacji przechowywanych danych.
- Jedyne co, to możemy modyfikować sam widok.
- Z tego powodu iterator i const_iterator dla string_view to ten sam typ.
- std::basic_string_view udostępnia interfejs będący w dużej mierze zmodyfikowanym podzbiorem tego z std::string.
- Przykład: string_view_2-4.cpp

std::string_view

- Inne metody: swap, copy, substr (ta nas będzie jeszcze interesować), compare, find, rfind, find_first_of, find_first_not_of, find_last_of, find_last_not_of
- Wszystkie metody są niemodyfikujące – co widac z pokazanych przykładów.

std::string_view

- Wracając do kwestii optymalizacji
- Dlaczego std::string_view ma prowadzić do optymalizacji, skoro udostępnia metody z pozoru takie same jak std::string.
- Gwarancja stałości danych pozwala nam mocno uprościć część przeprowadzanych operacji – przede wszystkim pozwala ograniczyć liczbę kopiowań.
- Najlepiej widać to na przykładzie metody substr()
- Przykład: string_view_substr.cpp

std::string_view - podsumowanie

- string_view można używać np. gdy mamy funkcję która przyjmuje const std::string& - wtedy jest duża szansa, że będzie można to zastąpić string_view, co pozwoli uprościć operacje wykonywane w tej funkcji.
- Więcej testów:
- https://www.bfilipek.com/2018/07/string-view-perf.html
- http://www.modernescpp.com/index.php/c-17-avoid-copying-with-std-string-view
- Warto zajrzeć, bo nie wszystkie są tak oczywiste jak ten przestawiony w przykładzie.
- string_view nie alokuje ani nie zarządza pamięcią
- string_view nie dodaje samemu znaku '\0' - jeśli dostanie tablicę bez niego, to będzie źle działał.

std::any

std::any

- Nowy typ dodany w c++17
- Jest to bezpieczny ze względu na typy (type-safe) „kontener” na jedną wartość dowolnego typu.
- Pochodzi z projektu Boost – jest tam typ o nazwie boost::any
- Jest to tzw. wrapper type - typ opakowujący jakiś obiekt – w naszym przypadku ten obiekt może mieć dowolny typ.
- Jeden z 3 nowych typów opakowujących – obok std::variant i std::optional

std::any

- Rozwinięcie idei void* w stronę bezpieczeństwa.
- std::any daje zdecydowanie więcej bezpieczeństwa od void* - nie możliwe jest np. rzutowanie na błędny typ – o błędzie zostaniemy poinformowani (np. za pomocą wyjątku).
- Przykład: void_star.cpp
- Przykład: any_basic_1.cpp

std::any

- std::any nie jest szablonem – w przeciwieństwie do optional i variant
- Aktualnie przechowywany typ = active type – ta sama nazwa powtórzy się jeszcze potem
- Domyślnie nie zawiera żadnej wartości
- Obecność wartości można sprawdzić za pomocą metody has_value()
- Można „zresetować” obiekt std::any za pomocą metody reset()
- Przy przypisywaniu nowego typu stary jest niszczony

std::any

- Dostęp do przechowywanej wartości za pomoca std::any_cast<T> - jest to szablon pozwalający nam podać typ docelowy
- Jeśli rzutowanie nie powiedzie się, to rzucany jest wyjątek std::bad_any_cast – co gwarantuje bezpieczeństwo pod względem typów – co było widać na pokazywanym wcześniej przykładzie.
- Możemy poznać przechowywany typ w czasie działania programu – używamy metody type(), zwraca std::type_info.
- Przykład: any_basic_2-3.cpp

std::any

- Zwiększone bezpieczeństwo względem void*, ale – zwykle jest tak, że zbiór możliwych do użycia typów jest w jakiś sposób ograniczony – to może jednak std::variant byłby lepszą opcją???
- Przydatne: std::make_any<>, metoda emplate i przypisanie.
- Przykład: any_basic_4.cpp

std::any

- Standard zachęca by implementacje std::any używały tzw. SBO – Small Buffer Optimization – czyli brak dynamicznej alokacji pamięci dla małych obiektów.

std::variant

std::variant

- Nowy typ dodany w C++17 – moim zdaniem jeden z ciekawszych
- Inspirowany typem variant z projektu Boost (jak wiele zmian w C++17).
- Nagłówek: <variant>
- Reprezentuje bezpieczną pod względem typów unię.

Krótka powtórka z unii

- Unia to specjalny rodzaj klasy, który może mieć w danym momencie aktywne tylko jedno ze swoich niestatycznych pól (data members) w danym czasie. Aktywne oznacza, że to wartość dla danego pola jest aktualnie przechowywana w pamięci.
- Zasadność stosowania: oszczędność pamięci – rozmiar jest możliwie minimalny – unia jest tak duża jak jej największe pole.
- Zastosowanie niskopoziomowe.

Krótka powtórka z unii 2

- union Nazwa_unii

```
{
    std::int32_t dw;
    std::uint16_t w;
    std::uint8_t b;
};
```

- Wygląda więc podobnie do zwykłej klasy czy struktury.
- Wymaga świadomości który typ jest aktualnie „aktywny”.
- Przykład: unions_before_cpp17_1.cpp

Krótka powtórka z unii 3

- Nie wchodząc w szczegóły – można zauważyć że wykorzystywanie unii w większych projektach może być problematyczne
- Pokazane zachowanie nie jest jedynym tego typu.
- Występujące problemy mogą często być rozwiązywalne, ale te rozwiązania nie zawsze są oczywiste.
- Zachowanie unii jest często zależne od implementacji (jak w pokazanym przykładzie).

Krótka powtórka z unii 4

- Inne ograniczenia unii: nie mogą mieć metod wirtualnych ani brać udziału w dziedziczeniu (nie mogą być klasą bazową ani pochodną).
- Największy problem pojawia się jednak, gdy pola unii to klasy z niestandardowymi (user defined) konstruktorami i destruktorami.
- Następuje wtedy komplikacja przy przełączaniu się pomiędzy aktywnymi polami.
- Przykład: unions_before_cpp17_2.cpp

Krótka powtórka z unii 5

- Dostaliśmy więc problem związany z czasem życia obiektów. Trzeba samodzielnie wywoływać konstruktory i destruktory jeśli są one nietrywialne.
- Ponadto w takim przypadku konstruktor i destruktor domyślny unii zostają wyłączone.
- Jest to kolejne utrudnienie zniechęcające do używania unii w większych, bardziej wysokopoziomowych projektach.
- Możliwe rozwiązanie – implementacja jakiejś klas opakowującej, z informacją o aktualnie przechowywanym typie.

std::variant

- Konieczność implementacji tego typu klasy znika wraz z C++17 (i wcześniej z projektem Boost) – pojawił się typ std::variant reprezentujący nową, bezpieczniejszą wersję unii.
- Rozwiązuje przedstawione wcześniej problemy w sposób niewidoczny dla użytkownika.
- Pozwala również poznać w dowolnym momencie aktualny typ aktywny.

std::variant

- template <typename … Types>

class variant;

- Pojęcie „sum types”
- Jako argumenty szablonu (variadic template) podajemy typy, które ma przechowywać variant.
- std::variant przechowuje obiekt bezpośrednio wewnątrz siebie (nie ma dynamicznej alokacji)
- W danej chwili jedna wartość, albo żadna (ale to w przypadku błedu).
- Przykład: variant_basic_1.cpp

std::variant

- Jeśli koniecznie chcemy mieć std::variant, który jest „”, to używamy std::monostate
- Jest to struktura pomocnicza reprezentująca pusty typ (stan).
- struct monostate { };
- Odpowiada to sytuacji, gdy std::variant nie ma pod danym indeksem żadnej wartości.
- Przydatne, gdy pierwszy typ nie ma konstruktora domyślnego – wtedy nie da się użyć konstruktora domyślnego dla std::variant (błąd kompilacji).

std::variant

- W takiej sytuacji na pierwszej pozycji umieszczamy właśnie std::monostate, który ma konstruktor domyślny.
- Przykład: variant_basic_2.cpp

- Pozyskiwanie informacji o aktualnie aktywnym typie i wartości można zrealizować w kilka sposobów.
- Przykład: variant_basic_3.cpp

std::variant

- 4 sposoby przypisania wartości do std::variant:
  - Operator przypisania
  - Metoda emplace (!)
  - Użycie std::get (pozyskanie referencji) lub podobnie z std::get_if
  - Visitor (ale o tym za chwilę)

Przykład: variant_change_value.cpp

std::variant

- Przewaga std::variant nad zwykłą unią oprócz możliwości pozyskania aktualnego indeksu i typu jest fakt, że w znacznie czytelniejszy sposób zarządza czasem życia przechowywanych obiektów – przy zmianie aktywnego typu wywoływane są odpowiednie konstruktory i destruktory.
- Przykład: variant_object_lifetime.cpp

std::variant i std::visit

- std::variant towarzyszy pomocny szablon funkcji – std::visit.
- Jego działanie polega na tym, że możemy wywołać jakąś funkcję (lub obiekt funkcyjny – ogólnie coś co można wywołać – callable) na wszystkich przekazanych do std::visit variantach.
- Uwaga: funkcja przekazana do std::visit jest tak naprawdę wywoływana raz, a jej argumenty to typy aktywne ze wszystkich przekazanych variantów – tzn. że musimy być przygotowanie na wszystkie sytuacje.

std::variant i std::visit

- Np. mamy dwa varianty: std::variant<int, double> i std::variant<std::string, int>
- Możliwe kombinacje argumentów wywołania (musimy przewidzieć wszystkie):

	int	double
std::string	int, std::string	double, std::string
int	int, int	double, int

std::variant i std::visit

- Template <typename Visitor, typename … Variants>

constexpr return_type visit(Visitor&& vis, Variants&& … vars);

- gdzie:
  - vis – funkcja, którą chcemy wywołać
  - vars – lista variantów, które chcemy przetworzyć za pomocą std::visit.
- Typ zwracany dedukowany na podstawie zwracanego wyrażenia (decltype) – musi być taki sam dla wszystkich kombinacji.
- Przykład: variant_visitor_1-2.cpp

std::variant i std::visit

- Zaleta – brak konieczności stosowania konstrukcji typu switch-case lub if do zdecydowania, której funkcji użyć.
- Działamy za pomocą przeładowywania funkcji – można też skorzystać z generic lambd (tak jak było to w przykładzie) – ma to jednak ograniczenia – nie zawsze każdy z typów ma ten sam interfejs.
- Zwykle dokonujemy wielokrotnego przeładowania operatora () - ma to dodatkową zaletę – pozwala zapamiętać stan.

std::variant i std::visit – nowe spojrzenie na polimorfizm

- std::variant wraz z std::visit dają nam nowy rodzaj polimorfizmu
- Zwykle gdy mówimy o polimorfizmie to mamy na myśli zbiór typów powiązanych ze sobą interfejsem (taki polimorfizm osiągamy za pomocą metod wirtualnych)
- Nowe podejście pozwala nam działać wspólnie na pozornie niezwiązanych ze sobą obiektach – posiadających jednak wspólne cechy pozwalające nimi operować w jednakowy sposób (nawet w ograniczonym stopniu).
- Przykład: variant_poli_1-2.cpp

std::variant - valueless_by_exception

- Wcześniej wspomniałem, że std::variant można doprowadzić do stanu, w którym nie będzie on miał żadnej wartości.
- Jest to oczywiście skutek wystąpienia błędu – jest kilka możliwości spowodowania takiego stanu, ja pokażę jedną:
- Przykład: variant_exception.cpp

std::optional

std::optional

- Nowy typ dodany w C++17
- Zaczerpnięty z projektu Boost (boost::optional)
- Nagłówek <optional>
- Template <typename T> class optional;

- Jest to typ opakowujący (wrapper) przechowujący opcjonalną wartość.
- Może ona istnieć, ale nie musi w danym momencie.

std::optional

- Przykładowe zastosowanie – funkcja, której wykonanie może się „nie udać” - wcześniej można było zwrócić z takiej funkcji parę std::pair<T, bool> albo jakiś kod błędu/ wartość specjalną (nullptr, -1).
- Podane rozwiązania nie są jednak czytelne, gdyż nie wyrażają w sposób jednoznaczny intencji twórcy programu.
- std::optional może w danej chwili znajdować się w jednym z dwóch stanów – może albo przechowywać wartość, albo nie.

std::optional

- Wartość przechowywana jest bezpośrednio wewnątrz optionala – nie ma dynamicznej alokacji pamięci.
- std::optional pochodzi ideowo z programowania funkcyjego
- Jest bezpieczny ze względu na typ.
- Powinno się go używać w sytuacjach, gdy fakt, że nie posiada on wartości jest równie naturalny jak fakt posiadania jej.
- Przykład: optional_basic_1.cpp

std::optional

- Zawiera wartość, gdy: został zainicjalizowany lub poddany przypisaniu za pomocą wartości typu T lub innego std::optional, który posiadał wartość
- Nie zawiera wartości, gdy: nie został zainicjalizowany lub przypisano mu wartość typu std::nullopt_t lub std::optional który nie zawiera wartości.
- Przykład: basic_optional_2.cpp

Operacje na std::optional

- emplace
- Przykład: optional_basic_3.cpp
- Porównania
- Przykład: optional_basic_4.cpp

Część II

invoke

std::invoke

- Służy ujednoliceniu wywoływania obiektów typu „callable”

- W przypadku std::invoke rozróżniamy trzy rodzaje obiektów typu „callable” :
  1. Wskaźnik na metodę klasy
  2. Wskaźnik na pole klasy
  3. Funkcja

- W zależności od tego do jakiej grupy należy obiekt który chcemy wykorzystać w std::invoke to proces wywoływanie się różni.

- std::invoke(f, t1, t2, …, tN);

Wskaźnik na metodę klasy T

- Jeżeli std::is_base_of<T, std::decay_t<decltype(t1)>>::value jest true, to invoke działa analogicznie do: **(t1.*f)(t2, …, tN)**
- Jeżeli std::decay_t<decltype(t1)> jest specjalizacją std::reference_wrapper, to invoke działa analogicznie do: **(t1.get().*f)(t2, …, tN)**
- W przeciwnym wypadku invoke działa analogicznie do: **((*t1).*f)(t2, …, tN)**

Wskaźnik na pole klasy T

- Jeżeli N == 1 oraz f jest wskaźnikiem na pole T:
  - Jeżeli std::is_base_of<T, std::decay_t<decltype(t1)>>::value jest true, to invoke działa analogicznie do: **t1.*f**
  - Jeżeli std::decay_t<decltype(t1)> jest specjalizacją std::reference_wrapper, to inoke działa analogicznie do: **t1.get().*f**
  - W przeciwnym wypadku invoke działa analogicznie do: **(*t1).*f**

Funkcja

- Gdy invoke nie spiełnia żadnego z poprzednich warunków, to działa analogicznie do:
  f(t1, t2, …, tN)

- Przykład std_invoke.cpp

apply

std::apply

- Pozwala na wywołanie obiektu „callable” z argumentami, które są przekazywane i wypakowywane z obiektu typu std::tuple, bądź obiektu, który częściowo implementuje interfejs std::tuple, czyli:
  - std::get
  - std::tuple_size
  - Np.: std::array, std::pair

- Przykład std_apply.cpp

make_from_tuple

std::make_from_tuple

- Podobnie jak std::apply pozwala na przekazanie argumentów w formie std::tuple, bądź klasy częściowo implementującej jej interfejs, natomiast w przeciwieństwie do std::apply służy do konstruowania obiektów z użyciem przekazanych parametrów. Wynikiem działania tej funkcji jest utworzony obiekt.

- Przykład std_make_from_tuple.cpp

for_each_n

std::for_each_n

- Podobnie jak std::for_each pozwala na zaaplikowanie obiektu funkcyjnego do obiektu który jest wynikiem dereferencji iteratora z przedziału od [first, first+n)

- Zapis:
  for_each_n(It first, Size n, Function f);

  for_each_n(ExecutionPolicy&& policy, It first, Size n, Function f);

- W pierwszym przypadku iterator musi spełniać założenia Input iteratora, a f musi być „move constructible”. Kolejność wykonania zostaje zachowana.

- W drugim przypadku iterator musi spełniać założenia Forward iteratora a f musi być „copy constructible”. Kolejnośc wykonania nie musi zostać zachowana.

- ~~Przykład for_each_n.cpp~~ brak wsparcia w kompilatorze

Execution policy

Polityki wykonania odnoszą się do zrównoleglania algorytmów. Wyróżniamy trzy polityki wykonania:

- sequenced_policy – nie zezwala na zrównoleglanie algorytmu
- parallel_policy – pozwala na zrównoleglenie algorytmu z pojedynczymi wykonaniami
- parallel_unsequenced_policy – pozwala na zrównoleglenie algorytmu oraz jego zwektoryzowanie

Różnica między par_unseq, a par

```
std::transform(
    // "Left" input sequence.
    x.begin(), x.end(),
    y.begin(), // "Right" input sequence.
    z.begin(),// Output sequence.
    mul);
```

```
load x[i]
load y[i]
mul
store into z[i]
```

```
load x[i... i+3]
load y[i...i+3]
mul // four elements at once
store into z[i...i+3]
```

Źródło: www.bfilipek.com/2017/08/cpp17-details-parallel.html

AGH

reduce

reduce

- Pozwala na wykonanie binarnej operacji zakresie zadanym przez iteratory

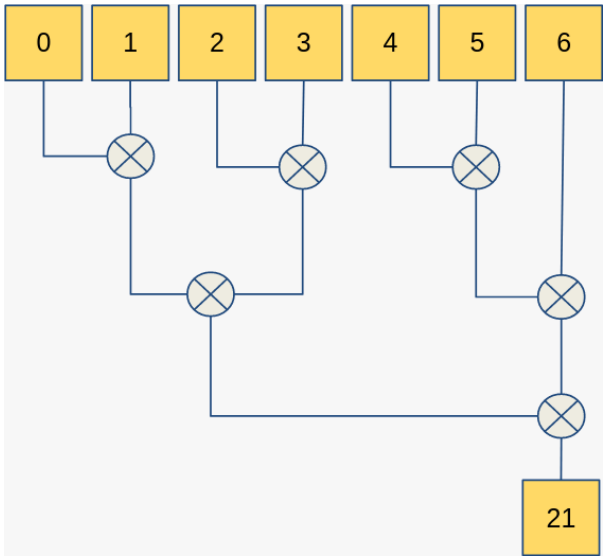
- Zapis:
  - reduce(InputIt first, InputIt last);
  - reduce(InputIt first, InputIt last, init);
  - reduce(InputIt first, InputIt last, init, binary_op_obj)

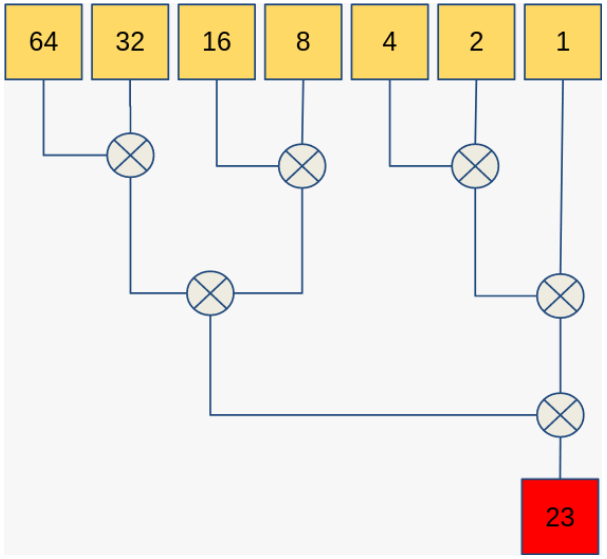
- ~~Przykład reduce.cpp~~

- Jest możliwe podanie execution policy jako pierwszego argumentu funkcji, natomiast wtedy iterator musi spełniać założenia forward iteratora.

reduce

- Wybierając współbieżne execution policy trzeba mieć na uwadze idące za tym konsekwencje





Źródło: https://blog.tartanllama.xyz/accumulate-vs-reduce/

- ~~Przykład reduce_par.cpp~~

exclusive_scan / inclusive_scan

exclusive_scan / inclusive_scan

- scan – operacja działająca na sekwencji obiektów. Wykonuje działanie na dwóch pierwszych elementach, następnie wynik tego działania jest wykonywany z następnym elementem

- Zapis analogiczny jak w przypadku reduce, nie występuje przypadek bez „init”.

- Kolejność wykonywania może być nie zachowana dla zrównoleglonego sposobu wykonania

Różnica między exclusive, a inclusive

exclusive

input numbers	1	2	3	4	5	6	...
prefix sums	0	1	3	6	10	15	21

inclusive

input numbers	1	2	3	4	5	6	...
prefix sums	1	3	6	10	15	21	...

- ~~Przykład scan.cpp~~

Pozostałe algorytmy wspierające execution policy

- for_each
- transform_reduce
- transform_exclusive_scan
- transform_inclusive_scan

try_emplace

try_emplace

- Używany przy std::map
- Działa jak emplace, jeżeli podany klucz w mapie nie istnieje
- W przeciwnym wypadku nie robi nic
- Przewagą try_emplace nad emplace jest to, że najpierw sprawdza, czy obiekt może zostać wstawiony, a dopiero później go tworzy

- Zapis:
  - try_empalce(key, …args)
  - try_emplace(hint, key, … args)

try_emplace

- W przypadku użycia bez „hinta” zwraca parę – iterator, która wskazuje na element o podanym kluczu oraz wartość logiczną która mówi nam, czy udało się wstawić nowy element.

- W przypadku użycia „hinta” funkcja szuka wolnego miejsca jak najbliżej przed tym iteratoram (hintem) i zwraca iterator na to miejsce (obiekt zostaje w tym miejscu utworzony).

- Przykład try_emplace.cpp

insert_or_assign

insert_or_assign

- Używany przy std::map
- Podobnie jak try_emplace występuje w wersji z hintem oraz bez
- Jeżeli element podany klucz nie istnieje w mapie, to wykonuje operację insert, w przeciwnym wypadku przypisuje elementowi znajdującemu się pod danym kluczem nową wartość
- Return analogiczny do try_emplace. W tym przypadku drugi element pary wskazuje na to, czy została wykonana insercja(true), czy tez przypisanie.
- Przykład insert_or_assign.cpp

Splicing for std::map

Splicing for std::list (poprzedni standard)

- Pozwala na tanie przenoszenie obiektów między listami
- Za pomocą splice możemy przenieść jeden, kilka lub nawet wszystkie elementy z jednej listy do drugiej
- Jest to wydajny sposób przenoszenia, ponieważ obiekty same w sobie nie są przenoszone, a jedynie obiekty zarządzające dostępem do elementów listy.
- Przykład list_splice.cpp

Splicing for std::map

- extract pozwala na tanie przenoszenie wybranych węzłów (klucz, wartość) między mapami

- Tak naprawdę obiekty nie są przenoszona, a jedynie obiekty „node handle”, które zarządzają dostępem do elementów, dzięki czemu operacja nie jest tak kosztowna.

Splicing for std::map

- merge pozwala na szybkie połączenie dwóch map
- Podobnie jak przy extract przenoszone są jedynie obiekty „node handle”
- Działanie polega na wywołaniu extract na każdym elemencie mapy która ma zostać połączona oraz na wywołoniu insert do mapy docelowej
- Przykład map_splicing.cpp

Źródła Część I

https://www.bfilipek.com/

https://en.cppreference.com/w/

https://arne-mertz.de/

https://pabloariasal.github.io/2018/06/26/std-variant/

http://www.modernescpp.com/index.php/c-17-avoid-copying-with-std-string-view

Źródła Część II

- https://en.cppreference.com
- http://open-std.org
- https://filipjaniszewski.com
- Code::Dive Conference
- www.bfilipek.com
- blog.tartanllama.xyz

Dziękujemy za uwagę