AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

**Praca inżynierska**

**Łukasz Czubala**

kierunek studiów: **informatyka stosowana**

# Porównanie wyrażeń lambda pomiędzy standardami języka C++

Opiekun: **dr inż. Janusz Malinowski**

**Kraków, luty 2020**

**Oświadczenie studenta**

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): ,,Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) ,,Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. --- Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

..... (czytelny podpis)

Merytoryczna ocena pracy przez opiekuna:

Merytoryczna ocena pracy przez recenzenta:

*Serdeczne podziękowania dla Pana dr. inż. Janusza Malinowskiego za wyrozumiałość oraz wsparcie podczas pisania niniejszej pracy.*

# Spis treści

# 1. Wstęp

Język C++ jest jednym z najpopularniejszych kompilowanych języków programowania. Pomimo dużej konkurencji ze strony nowszych i zarazem prostszych w użyciu języków programowania, przede wszystkim interpretowanych, wciąż cieszy się dużą popularnością.

Głównymi powodami dla którego nie został wyparty są jego wydajność oraz niskie zużycie pamięci. Mimo iż większość kodu w dzisiejszych aplikacjach pisana jest w innych językach, to najbardziej krytyczne fragmenty aplikacji tworzone są z pomocą języka C++.

Bezsprzeczną rewolucją w programowaniu w tym języku było opublikowanie standardu C++11 we wrześniu 2011 r. Jedną z wyczekiwanych zmian było wprowadzenie obsługi wyrażeń lambda, które są odpowiednikiem funkcji anonimowych w innych językach. Jest to szczególnie przydatne w kontekście upraszczania zapisu, oraz gdy funkcji będziemy używać ograniczoną liczbę razy wiedząc że użycie funkcji anonimowej może być wygodniejsze od użycia funkcji nazwanej.

Stały się one powszechnie używane, także dzięki bibliotece standardowej, której zestaw klas oraz interfejsów znacząco rozszerza język C++. Algorytmy standardowe wymagają tworzenia funktorów, na przykład w algorytmie sortującym „std::sort”. Właśnie tutaj z pomocą przyjdą nam funkcje anonimowe.

Celem pracy inżynierskiej było opisanie zmian jakie miały miejsce w poszczególnych standardach wraz z zapowiedzianymi zmianami w nadchodzącym standardzie C++20, transformacja wyrażeń lambda do kodu widzianego z perspektywy kompilatora oraz porównanie wydajności kodu z wyrażeniami lambda w zależności od użytego standardu.

# 2. C++03 - typy funkcyjne i funktory

Korzystając ze standardu języka C++ wydanego w 2003 roku, czyli ISO/IEC 14882:2003 [1] praca programisty nie była zbyt prosta i wygodna. Powodów na wykorzystanie algorytmów takich jak np. std::remove_if, std::find_if, std::sort było mnóstwo. Aby z nich korzystać, potrzebne było użycie jakiegoś kryterium, aby wiedzieć co zrobić z kolejnymi elementami kontenerów zapewnianych przez bibliotekę.

## 2.1. Przykład funktora dla algorytmu STL

Takim kryterium mógł być dowolny obiekt możliwy do wywołania, który następnie wywoływano na tych elementach. Tymczasem programista miał do wyboru tylko typ funkcyjny w postaci wskaźnika na funkcję lub funktor, czyli obiekt funkcyjny, który może być wywoływany jak zwykła funkcja. Funktorem może zostać każda klasa, posiadająca zdefiniowany operator().

Na przykład:

```
#include <iostream>
#include <algorithm>
#include <vector>

struct WypiszFunktor {
   void operator()(unsigned int x) const {
      std::cout << x << std::endl;
   }
};

int main() {
   std::vector<unsigned int> v;
   v.push_back(1);
   v.push_back(2);
   std::for_each(v.begin(), v.end(), WypiszFunktor());
}
```

## 2.2. Funktor w innym zasięgu

Problem pojawia się jednak wtedy, gdy trzeba napisać osobną funkcję lub funktor w innym zakresie niż wywołanie algorytmu. Rozwiązanie, które wydaje się najbardziej naturalne, to napisanie lokalnej klasy funktorów, która jest obsługiwana w C++03.

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    struct WypiszFunktor {
        void operator()(unsigned int x) const {
            std::cout << x << std::endl;
        }
    };

    std::vector<unsigned int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), WypiszFunktor());
}
```

Próba kompilacji kodu na GCC (GNU Compiler Collection) w wersji 5.5.0 z flagą -std=c++98 kończy się niepowodzeniem. Otrzymujemy następujący komunikat błędu:

```
error: template argument for
'template<class _IIter, class _Funct> _Funct
std::for_each(_IIter, _IIter, _Funct)'
uses local type ‘main()::WypiszFunktor’
```

Dzieje się tak dlatego, iż w C++03 nie można utworzyć instancji szablonu z typem lokalnym. Zakres musi się pokrywać z widocznością konkretyzacji szablonu. By tak było, obiekt musiałby być globalny.

Między innymi z powodu powyższych ograniczeń Komitet Standaryzacyjny C++ (ISO/IEC JTC1/SC22/WG21) [2] zaczął projektować nową funkcjonalność, dzięki której moglibyśmy stworzyć coś w miejscu użycia [3]. Tak powstały wyrażenia lambda, nazywane również funkcjami lambda.

# 3. C++11 – wyrażenia lambda

Standard języka ISO/IEC 14882:2011 (C++11 jest nieformalną nazwą) został opublikowany w 2011 roku [4]. Poza nowymi możliwościami oraz modyfikacjami obejmującymi rdzeń języka oraz bibliotekę standardową znalazło się również miejsce dla wyrażeń lambda.

Funkcjonalność została dodana używając nowej składni. Kompilator rozwija ją do prawdziwej klasy. Takie podejście powoduje, iż mamy dostęp do zalet jak i wad silnej typizacji.

Na początku, pozwolę sobie przytoczyć kilka definicji z ostatecznego szkicu standardu C++11 [5], aby przedstawić podstawowe pojęcia.

[expr.prim.lambda#1]:
*"Wyrażenia lambda zapewniają zwięzły sposób tworzenia prostych obiektów funkcyjnych"* (Lambda expressions provide a concise way to create simple function objects) [6].

[expr.prim.lambda#2]:
"*Rozwinięcie wyrażenia lambda prowadzi do tymczasowego prvalue"* (The evaluation of a lambda-expression results in a prvalue temporary)
*"Ten obiekt tymczasowy jest nazywany obiektem domknięcia"* (This temporary is called the closure object.)
*"Uwaga: Obiekt domknięcia zachowuje się jak obiekt funkcyjny"* (Note: A closure object behaves like a function object) [7]

[expr.prim.lambda#3]:
*"Typ wyrażenia lambda (który jest również typem obiektu domknięcia) jest unikalnym, nienazwanym niezwiązkowym typem klasy - nazywanym typem domknięcia"* (The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type — called the closure type.) [8]

## 3.1. Typy wyrażeń lambda

Typ domknięcia jest generowany w sposób unikalny poprzez kompilator. Dlatego też nie istnieje sposób, dzięki któremu moglibyśmy poznać nazwę lambdy.
Skutkuje to tym, iż chcąc wydedukować jej typ, powinniśmy skorzystać ze słowa kluczowego "auto", tak jak w poniższym zapisie:

```
auto przykladLambdy = [] (int x) { std::cout << x << std::endl; }
```

Typ domknięcia powiązany z wyrażeniem lambda ma usunięty konstruktor domyślny oraz operator przypisania. Ma również niejawnie zadeklarowany konstruktor kopiujący i może mieć niejawnie zadeklarowany konstruktor przenoszący. Trzeba tutaj także wspomnieć, iż konstruktory są domyślnie zdefiniowane w taki sam sposób, jak każdy inny niejawnie zadeklarowany konstruktor kopiujący lub przenoszący [9].

Powyższe własności powodują, że próba kompilacji poniższego kodu zakończy się niepowodzeniem:

```
auto przykladLambdy2 = [&x]() { ++x; };
decltype(przykladLambdy2) kopiaLambdy2;
```

Z kompilatora GCC otrzymamy następujący komunikat o błędzie:

```
error: use of deleted function 'main()::<lambda()>::<lambda>()'
 decltype(przykladLambdy2) kopiaLambdy2;
                           ^~~~~~~~~~~~
note: a lambda closure type has a deleted default constructor
```

## 3.2. Operator wywoływania

Kod, który umieszczamy w ciele wyrażenia lambda poddawany jest przekształceniom w taki sposób, aby użyć poprawnego typu domknięcia. Znajduje on swoje miejsce wewnątrz metody "operator()".
Kompilator używa słowa kluczowego "inline" przy deklaracji operatora wywołania [10]. Oznacza to, iż w miejscu wywołania wstawiany jest kod, zamiast wskaźnika do funkcji [11].

## 3.3. Słowo kluczowe mutable

Operator wywołania domyślnie jest tworzony z modyfikatorem "const" po liście argumentów, który jawnie informuje kompilator o tym, że metoda nie modyfikuje składowych [12].
Aby to zmienić, możemy zamieścić specyfikator "mutable" za listą argumentów w następujący sposób:

```
auto przykladLambdy3 = [](int x) mutable { ++x; };
```

Warto jednak zauważyć, że w tym przypadku kod zadziała również bez modyfikatora "mutable", ponieważ działamy na zwykłej kopii parametru nie przechwytywanej z zewnątrz, do kodu wewnątrz lambdy.

Trzeba to rozumieć w ten sposób, iż modyfikator "const", w przypadku wyrażeń lambda, informuje jedynie o braku możliwości modyfikacji tylko tych zmiennych, które przechwytywane są do ciała lambdy za pomocą "operator []" przez wartość. Dlatego też poniższy kod, wywołany bez "mutable":

```
auto przykladLambdy4 = [a]() { a++; };
```

Zakończy się wypisaniem następującego błędu z kompilatora GCC:

```
In lambda function:
error: increment of read-only variable 'a'
 auto przykladLambdy4 = [a](){ a++; };
                                  ^~
```

## 3.4. Przechwytywanie operatorem [ ]

W konstrukcji wyrażenia lambda występuje "operator []" (nazywany też w tym kontekście „klauzulą przechwytywania” (capture clause). Wewnątrz nawiasów kwadratowych możemy zawrzeć elementy, które lambda powinna przechwycić. Do wyboru mamy kilka opcji.

### Puste nawiasy

Użycie tej konstrukcji oznacza, iż wewnątrz lambdy nie będzie można użyć żadnej nazwy z otaczającego kontekstu.

```
auto lambda = [](){ std::cout << "Tylko wypisuję tekst" << std::endl; };
```

## Niejawne przechwycenie przez referencję

W tym przypadku lambda będzie mieć dostęp do zapisu i odczytu wszystkich zmiennych automatycznych, osiągalnych z zakresu, w którym została utworzona. Obiekt domknięcia (closure), czyli instancja klasy domknięcia będzie przechowywać referencje do zewnętrznych zmiennych. Oznacza to, iż po przechwyceniu zmiennych można korzystać z nich wewnątrz ciała lambdy.

```
unsigned int a = 1;
auto lambda = [&](){ std::cout << "Mogę zmienić a=" << ++a << std::endl; };
```

## Niejawne przechwycenie przez wartość

Wariant ten jest podobny do przechwycenia przez referencję. Również tutaj lambda będzie mieć dostęp do nazw zmiennych z zewnętrznego kontekstu, w którym została utworzona. Różnicą jest to, iż te nazwy będą się odnosić do lokalnych kopii przechowywanych przez instancję klasy domknięcia. Wartości będą takie, jak w momencie tworzenia obiektu domknięcia. Zmiana wartości zmiennej wewnątrz ciała lambdy nie będzie widoczna dla zewnętrznego kontekstu.

```
unsigned int b = 1;
auto lambda = [=](){ std::cout << "Mogę wypisać b=" << b << std::endl; };
```

## Jawne przechwycenie zmiennych z listy przechwytywania

Użycie tego sposobu (czyli "[capture-list]") pozwala na przechwytywanie zmiennych zarówno przez wartość, jak i przez referencję. Zmienne wymienione z poprzedzeniem ich operatorem pobrania adresu (operator &) zostaną przechwycone przez referencję, w innym przypadku, zostaną przechwycone przez wartość.

### Niejawne przechwycenie przez referencję oraz capture-list

W tej konstrukcji (czyli „[&, capture-list]) przez referencję zostaną przechwycone wszystkie zmienne poza wymienionymi na liście. Te z listy będą przechwytywane przez wartość. Na liście możemy umieścić wskaźnik „this” (czyli wskaźnik do obiektu na którym w danym kontekście wywoływana jest metoda).

### Niejawne przechwycenie przez wartość oraz capture-list

Ostatnia kombinacja (czyli „[=, capture-list]”) pozwala w sposób niejawny na przechwycenie wszystkich zmiennych przez wartość poza wymienionymi na liście i poprzedzonymi operatorem „&”, które to zostaną przechwycone przez referencję. Lista nie może w sobie zawierać wskaźnika „this”.

## 3.5. Przechwytywanie zmiennych globalnych

Próba przechwycenia zmiennej globalnej przez wartość w sposób niejawny (czyli „[=]”) może wprowadzić programistę w błąd. Moglibyśmy się spodziewać, że taka zmienna zostanie przechwycona przez wartość, jednak tylko zmienne automatyczne mogą zostać przechwycone. Modyfikacje na niej wewnątrz lambdy nie będą więc zmianą na kopii. Kompilator GCC ostrzeże nas jedynie, gdy będziemy przechwytywali taką zmienną jawnie, w następujący sposób:

warning: capture **of** variable 'zmienna_globalna' **with** non-automatic storage duration

## 3.6. Przechwytywanie zmiennych statycznych i stałych

Zmienne statyczne zachowują się przy próbie przechwytywania podobnie do zmiennych globalnych. Operacje wewnątrz lambdy będą wykonywane bezpośrednio

na zmiennej statycznej, nie na jej kopii. Tak samo jak dla zmiennych globalnych, kompilator GCC ostrzeże nas tylko przechwytywaniu zmiennej statycznej jawnie.

Jeżeli przechwycimy stałą (np. „const int i = 1”), to nadal pozostanie one niezmienna. „Stałość” zostanie zachowana. Próba kompilacji kodu ze zmianą wartości stałej w wyrażeniu lambda kończy się identycznym błędem, co dla zwykłej funkcji. Zostaje wyświetlony błąd o próbie modyfikacji zmiennej tylko do odczytu.

## 3.7. Przechwytywanie składników klasy

Przykład przechwytywanie składników (właściwości) klasy został zaprezentowany w poniższym kodzie:

```
struct A
{
   std::function<void()> f()
   {
      return [=] { std::cout << napis << std::endl; };
   }

   std::string napis;
};

int main()
{
  auto lambda1 = A{"tekst1"}.f();
  auto lambda2 = A{"tekst2"}.f();
  lambda1();
  lambda2();
}
```

Kod wydaje się poprawny, jednak po dokładnym przeanalizowaniu, nie jesteśmy w stanie powiedzieć co dokładnie wydarzy się, gdy wywołamy ostatnie dwie linie, ponieważ używamy obiektów tymczasowych. Jest to „zachowanie niezdefiniowane” [13], a problem jest nazwany jako „problem wiszącej referencji” (dangling reference problem). Nie możemy przechwycić również składnika przez kopię, ponieważ kod nie skompiluje się.

Z pomocą przychodzą nam dopiero nowsze standardy języka. Kompilując ten przykład przy użyciu GCC w wersji 9.2.0 z flagą -std=c++2a, zostanie wyświetlone ostrzeżenie oraz porada, aby w wyrażeniu lambda przechwycić „*this” w sposób jawny. Rozwiązanie to jest też poprawne dla standardu C++17, jednak nie zostaniemy ostrzeżeni przed popełnieniem błędu i zachowaniem niezdefiniowanym. W C++14 natomiast problem można obejść korzystając z inicjalizacji przechwytywanej zmiennej.

## 3.8. Obiekty typu MoveConstructible i MoveAssignable

Działanie na obiekcie, który jest MoveConstructible oraz MoveAssignable [14] (potocznie „Move-only”, przykładem jest unique_ptr), w standardzie C++11, nie pozwala na przechwycenie zmiennej przez wartość. Taki kod nie zostanie skompilowany. Można ją przechwycić jedynie przez referencję. Takie użycie może jednak prowadzić do problemów, ponieważ nie są wtedy przenoszone prawa własności do zmiennej.

## 3.9. Określanie zwracanego typu

W wyrażeniu lambda, zaraz za listą argumentów, można jawnie określić zwracany typ. W tym celu należy użyć następującej konstrukcji:

```
[]() -> Typ { }
```

Jeśli typ zwracany nie jest określony, zostanie wydedukowany przez kompilator. Automatyczna dedukcja nie powiedzie się, gdy użyjemy wielu wyrażeń „return” zwracających różne typy (np. „double” i „int”). Wtedy warto określić typ zwracany.

## 3.10. IIFE - Immediately Invoked Function Expression

IIFE [15], czyli “natychmiastowe wywołanie funkcji” jest często używanym narzędziem w języku JavaScript.
Tą technikę można wdrożyć również w języku C++, choć z zupełnie innych powodów. Dzięki IIFE możemy uzyskać bardziej czytelny kod, kiedy inicjalizacja obiektu stałego jest skomplikowana. Zanim było to możliwe, częstym wyborem programisty była rezygnacja ze specyfikatora „const”.
Natychmiastowe wywołanie wyrażenia lambda przy użyciu operatora() wraz z inicjalizacją obiektu stałego wygląda następująco:

```
const auto val = []() { /* skomplikowany kod */ }();
```

## 3.11. Konwersja do wskaźnika na funkcję

Typ domknięcia dla wyrażeń lambda z pustą listą przechwytywania, posiada niewirtualną, niejawną, stałą funkcję konwertującą do wskaźnika na funkcję, mającą takie same parametry i typ zwracany, co operator wywołania funkcji typu domknięcia. Wartością zwracaną przez tą funkcję konwertującą jest adres funkcji, która po wywołaniu ma taki sam efekt, jak inwokacja operatora wywołania typu domknięcia [16]. Przykład konwersji wyrażenia lambda do wskaźnika na funkcję został zaprezentowany w poniższym kodzie:

```
void (*funkcja)(int) = [](int x) { std::cout << x << std::endl; };
```

# 4. C++14 – ulepszenia przechwytywania i inicjalizacji

Standard języka ISO/IEC 14882:2014 (C++14 jest nieformalną nazwą) został opublikowany w 2014 roku [17]. Głównym postawionym przed nim celem była praca nad poprawkami oraz rozszerzeniem funkcjonalności wprowadzonych przez poprzedni standard.

Prace nad tym wydaniem trwały dużo krócej, niż nad poprzednim, tylko trzy lata. Intencją zmiany podejścia do cyklu wydawniczego była pomoc twórcom kompilatorów. Ma to owocować szybszemu wprowadzaniu nowości dla końcowych użytkowników.

Zmiany polegały przede wszystkim na rozwiązaniu kilku problemów, na które uskarżali się użytkownicy korzystając ze standardu C++11.

## 4.1. Dedukcja zwracanego typu

C++14 wprowadził wiele drobnych poprawek, które przekładają się na lepszą dedukcję typów w funkcjach. Zmiany te zostały wprowadzone jednocześnie do wyrażeń lambda.

Typ zwracany przez wyrażenie lambda jest „auto”. Zastępowany jest przez „*opóźnioną deklarację typu zwracanego”* (trailing-return-type) [18].

W poniższym przykładzie typem zwracanym będzie „int&”

```
int j = 2;
auto lambda = [&]() -> auto&& { return j; };
```

## 4.2. Przechwytywanie z inicjalizatorem

W nowym standardzie można inicjalizować nowe zmienne w obiekcie domknięcia, bez potrzeby istnienia tych zmiennych w otaczającym zakresie. Inicjalizacja może być wyrażona przez dowolne wyrażenie. Typ nowej zmiennej jest dedukowany na podstawie typu tworzonego przez wyrażenie.

```
int main() {
        int a = 1;
        int b = 2;
        auto lambda = [c = a + b]() { std::cout << c << std::endl; };
        lambda();
}
```

W powyższym przykładzie zmienna „c” będzie typu „int”.

## 4.3. Move

Przechwytywanie z inicjalizatorem rozwiązuje również problem ze zmiennymi „move-only” (jak np. „std::unique_ptr”). Można przesuwać obiekty do składnika obiektu domknięcia.

```
int main()
{
   std::unique_ptr<int> liczba(new int{10});
   auto lambda = [ptr=std::move(liczba)] { std::cout << *ptr << std::endl; };
   lambda();
}
```

Należy w tym miejscu pamiętać, iż próba wykonania operacji na przesuniętym już obiekcie spowoduje „Naruszenie ochrony pamięci” [19].

## 4.4. Optymalizacja poprzez inicjalizator

Kolejnym atutem inicjalizatora jest możliwość obliczenia jakiejś wartości przy jego pomocy jeden raz, zamiast przy każdym wywołaniu wyrażenia lambda.

```
int main()
{
    using namespace std::string_literals;
    std::vector<std::string> v = {"a", "b", "c", "ab", "ac", "bc"};
    auto it1 = std::find_if(v.begin(), v.end(),
                            [](std::string const& s) { return s == "a"s + "b"s; });
    auto it2 = std::find_if(v.begin(), v.end(),
                            [obl="a"s + "b"s](std::string const& s) { return s == obl; });
}
```

## 4.5. Przechwytywanie składników klasy

Inicjalizację można wykorzystać także do przechwytywania składników klasy. Jest to kolejny sposób na rozwiązanie „problemu wiszącej referencji” w poniższym przypadku. Przechwytywanie niejawne przez wartość prowadzi do zachowania niezdefiniowanego. Z tego powodu bezpieczniej jest przechwycić kopię składnika.

```
struct A
{
    std::function<void()> f()
    {
        return [napis = napis] { std::cout << napis << std::endl; };
    }

    std::string napis;
};

int main()
{
   auto lambda1 = A{"tekst1"}.f();
   auto lambda2 = A{"tekst2"}.f();
   lambda1();
   lambda2();
}
```

Wyrażenie lambda wewnątrz funkcji przechwytuje składnik (właściwość) klasy do obiektu domknięcia poprzez kopiowanie. Ponadto deklarując funkcję, można skorzystać ze słowa kluczowego „auto”, aby dokonać dedukcji typu całej funkcji, zastępując „std::function<void()>”.

## 4.6. Lambdy generyczne

W C++11 deklaracja parametrów do wyrażeń lambda musi być dokonana z użyciem konkretnego typu. C++14 pozwala na zadeklarowanie parametrów przy pomocy słowa kluczowego „auto”. Mechanizm ten nazywany jest „Generic lambda” [20]

```
auto lambda = [](auto a, auto b) { return a + b; }
```

Dedukcja typu parametru wyrażenia lambda następuje w ten sam sposób, jak dedukcja typu argumentu szablonu. Generowany przez kompilator kod jest analogiczny do poniższego przykładu deklaracji szablonowej:

```
struct KlasaDomkniecia
{
   template <typename T1, typename T2>
   auto operator()(T1 a, T2 b) const
   {
      return a + b;
   }
};

auto lambda = KlasaDomkniecia();
```

# 5. C++17 – constexpr i wskaźnik *this

Standard języka ISO/IEC 14882:2017 (C++17 jest nieformalną nazwą) został opublikowany w 2017 roku [21]. W kontekście wyrażeń lambda, Komitet Standaryzacyjny skupił się w tym wydaniu przede wszystkim na poprawieniu wydajności. Nie zabrakło również miejsca dla poprawy bezpieczeństwa użycia funkcji anonimowych wewnątrz metod klas.

## 5.1. Lambdy constexpr

Specyfikator „constexpr” (czyli „uogólnione wyrażenia stałe”) to funkcjonalność dodana jeszcze w C++11. Jego głównym zadaniem jest poprawa wydajności kodu poprzez wykonywanie możliwych obliczeń w czasie kompilacji, zamiast w czasie wykonania.

Użycie tego specyfikatora mówi również, iż wartość zmiennej, obiektu, funkcji bądź metody może być użyta w innych stałych wyrażeniach, ponieważ jest stała.

Istnieje rozbudowana lista wymagań, aby funkcja lub metoda mogła być wyrażeniem stałym [22]. Niektóre z nich to:
- funkcja nie może być wirtualna
- musi zwracać typ inny niż „void”, będący literałem
- każdy z parametrów musi być literałem
- nie może definiować nowych typów danych, ani deklarować żadnych zmiennych statycznych
- ciało funkcji nie może zawierać instrukcji skoku, bloków obsługi wyjątków, statycznej deklaracji potwierdzenia

Począwszy od C++17, spełnienie wszystkich postawionych wymagań przez „operator()” w wyrażeniu lambda powoduje, iż jest on definiowany jako „constexpr” [23], a sama lambda staje się literałem. W takiej lambdzie, można korzystać z innych stałych wyrażeń.

Często optymalizowane mogą to być skomplikowane kalkulacje. W poniższym przykładzie wyraz Ciągu Fibonacciego zostanie obliczony w czasie kompilacji.

```
constexpr long int fibonacci(int n)
{
   return (n <= 1)? n : fibonacci(n-1) + fibonacci(n-2);
}

int main ()
{
   const long int wynik = fibonacci(30);
   std::cout << wynik;
}
```

## 5.2. Przechwytywanie *this

Standard C++17 wprowadza funkcjonalność, która pozwala przede wszystkim na zapobieganiu problemowi wiszącej referencji, występującemu przy przechwytywaniu „this” jako wskaźnik, gdyż obiekt na który wskazujemy może nie istnieć w momencie wywołania lambdy.

Używając w klauzuli przechwytywania „*this” obiekt domknięcia (anonimowy obiekt funkcyjny) jest w całości kopiowany do każdej lokacji wywołań, gdzie wyrażenie lambda jest wywoływane [24]. Jest to szczególnie zalecane podczas używania tych wyrażeń w programach wielowątkowych [25].

Poniższy przykład jest w pełni bezpieczny od problemu wiszącej referencji.

```
struct A
{
    std::function<void()> f()
    {
        return [*this] { std::cout << napis << std::endl; };
    }

    std::string napis;
};

int main()
{
   auto lambda1 = A{"tekst1"}.f();
   auto lambda2 = A{"tekst2"}.f();
   lambda1();
   lambda2();
}
```

Niezależnie od użytego kompilatora zachowanie jest ściśle zdefiniowane. W sposób poprawny zostanie najpierw wypisany „tekst1”, a następie „tekst2”. Gdy z powodu przechwycenia niejawnego przez kopię występowało zachowanie niezdefiniowane, kod skompilowany i zoptymalizowany za pomocą GCC wypisywał na wyjście dwukrotnie „tekst2”.

# 6. C++20 – nadchodzące zmiany

Standard języka, który jest potocznie nazywany jako C++20 lub C++2a, ma nazwę roboczą ISO/IEC CD 14882 [26]. Wydanie standardu jest zaplanowane do końca lutego 2020 r. [27].

Wiele zapowiedzianych zmian polega na doprecyzowaniu funkcjonalności wyrażeń lambda w bardziej zaawansowanych przypadkach użycia, aby twórcy kompilatorów mogli ostrzegać użytkowników o możliwych problemach. Jedną z bardziej zauważalnych różnic, będą lambdy szablonowe.

## Wyraźne przechwytywanie this

Zmiana polega na zezwoleniu w klauzuli przechwytywania na „[=, this]”. Zdaniem osób przedstawiających tę propozycję, pożądane jest zwiększenie czytelności kodu poprzez wyraźne określanie rodzaju przechwytywania wskaźnika „this”, bądź kopii obiektu [28].

Podczas użycia „[=]” wewnątrz metody, zostaniemy przez kompilator poinformowani, iż takie użycie jest „przestarzałe” (deprecated), czyli odradzane.

## Szablonowe lambdy

Propozycja polega na dodaniu do standardu możliwości programowania uogólnionego na wyrażeniach lambda z użyciem szablonów [29]. Na liście parametrów szablonu będzie można na przykład określić typ elementów konkretnego kontenera.

```
auto lambda = []<typename T>(std::vector<T> t){};
```

## Inne poprawki

Wśród innych proponowanych nowości dotyczących lambd warto wymienić:

- Zmienna lista argumentów w klauzuli przechwytywania [30]
- Przechwytywanie powiązań strukturalnych [31]
- Domyślnie konstruowalne i możliwe do przypisania bezstanowe lambdy [32]
- Wyrażenia lambda w kontekście nie poddanym ewaluacji [33]
- Uproszczenia przechwytywania domniemanego [34]

# 7. Wyrażenia lambda z perspektywy kompilatora

Aby w przybliżeniu zobaczyć, w jaki sposób kompilator rozwija kod programu zawierającego wyrażenie lambda, można użyć narzędzia o nazwie *C++ Insights*. Dokonuje ono transformacji kodu w taki sposób, aby rzeczy, które są celowo ukryte przed użytkownikiem były bardziej widoczne [35].

## Przykład z pustą listą przechwytywania

Następujący kod zawiera proste wyrażenie lambda.

```
int main()
{
   std::vector<int> v;
   v.push_back(1);
   v.push_back(2);
   std::for_each(v.begin(), v.end(), [] (int x) { std::cout << x << std::endl; });
}
```

Algorytm std::for_each uruchamia funkcję anonimową, która wypisuje element na domyślne wyjście dla dwóch elementów kontenera.

Po uruchomieniu narzędzia, zostaje przekształcony do następującej postaci:

```
int main()
{
  std::vector<int> v = std::vector<int, std::allocator<int> >();
  v.push_back(1);
  v.push_back(2);

  class __lambda_10_39
  {
    public:
    inline /* constexpr */ void operator()(int x) const
    {
      std::cout.operator<<(x).operator<<(std::endl);
    }

    using retType_10_39 = void (*)(int);
    inline /* constexpr */ operator retType_10_39 () const noexcept
    {
      return __invoke;
    };

    private:
    static inline void __invoke(int x)
    {
      std::cout.operator<<(x).operator<<(std::endl);
    }

    public:
    // inline /*constexpr */ __lambda_10_39(__lambda_10_39 &&) noexcept = default;

  };

  std::for_each(__gnu_cxx::__normal_iterator<int *,
                                  std::vector<int, std::allocator<int> > >(v.begin()),
                __gnu_cxx::__normal_iterator<int *,
                                  std::vector<int, std::allocator<int> > >(v.end()),
                __lambda_10_39(__lambda_10_39{}));
}
```

Analizując rozwinięty kod, można zauważyć, że wyrażenie lambda to tak naprawdę klasa o unikalnej nazwie wygenerowanej przez kompilator (w tym przypadku typ to „__lambda_10_39”). Zawiera operator() pozwalający na wywołanie bezpośrednie. W celu optymalizacji wszystkie metody są „inline”, a gdy jest to możliwe, metody publiczne definiowane są jako „constexpr”.

Z powodu pustej listy przechwytywania, lambda może być zapisana jako wskaźnik na funkcję. Wyjaśnia to obecność „retType_10_39” oraz statycznego wywoływacza „__invoke”. Jest on statyczny, aby była możliwość użycia tej funkcji anonimowej bez istnienia obiektu klasy.

## Przykład z niepustą listą przechwytywania

Inny wynik przekształcenia wyrażenia lambda uzyskamy dla przykładu z niepustą klauzulą przechwytywania:

```
int main()
{
  int a = 2;
  auto funkcja = [&a](const int b) { a+=b;};
  funkcja(1);
}
```

Przetworzony kod prezentuje się następująco:

```
int main()
{
  int a = 2;

  class __lambda_6_17
  {
    public:
    inline /*constexpr */ void operator()(const int b) const
    {
      a += b;
    }

    private:
    int & a;

    public:
    __lambda_6_17(int & _a)
    : a{_a}
    {}

  };

  __lambda_6_17 funkcja = __lambda_6_17{a};
  funkcja.operator()(1);
}
```

Analizując przekształcony kod, można zauważyć, że przechwycenie referencji odbywa się za pomocą konstruktora klasy „__lambda_6_17”. Tutaj również „operator()” jest „inline”, ponadto specyfikator „constexpr” zostanie zastosowany, gdy będzie to możliwe. Prywatnym składnikiem klasy jest referencja na zmienną „a”, zapisywana podczas inicjalizacji. Słowo kluczowe „auto” w funkcji „main” zostało rozwinięte jako typ lambdy, a wywołanie jej to tak naprawdę wywołanie operatora nawiasu.

Z powodu niepustej listy przechwytywania, nie ma możliwość zapisania tej funkcji anonimowej w postaci wskaźnika na funkcję.

# 8. Środowisko testowe

Do celu uruchomienia przetestowanych programów użyto dystrybucji systemu operacyjnego „Ubuntu 16.04.6 LTS” z rodziny „GNU/Linux”. Do skompilowania programów użyto kompilatora GCC w wersji 9.2.1.

W celu zapewnienia jednolitego sposobu pomiaru czasu w stosunku do określonego fragmentu kodu, a nie działania całego programu użyto funkcjonalności o nazwie Boost.Chrono [36] z kolekcji bibliotek programistycznych Boost, w wersji „boost 1.58.0”. Funkcjonalność ta pozwala na odmierzenie odstępu czasu pomiędzy dwoma wybranymi punktami w prosty sposób. Dzięki temu możemy obliczyć różnicę pomiędzy nimi. Oferowana dokładność pomiaru czasu (w sekundach) to sześć miejsc po przecinku, czyli w mikrosekundach.

Aby możliwe było użycie Boost.Chrono przy użyciu GCC należało użyć flag „-lboost_system -lboost_timer -lboost_chrono -lrt”. Flagę optymalizacji ustawiono na poziomie pierwszym.

# 9. Zrealizowane scenariusze testowe

W ramach testów wydajności przygotowano dwa scenariusze. Przed testami postawiono warunek, aby była to jak najbardziej zbliżona funkcjonalność pomiędzy standardami języka C++, przy zachowaniu pełnej staranności pod względem wydajności programu. Celem takiego założenia była możliwość porównania wszystkich standardów, mimo przeszkody, iż wielu funkcjonalności języka nie ma w starszych wersjach standardu. Aby spełnić to założenie, autor dołożył starań, aby kod był w zadanym kontekście maksymalnie wydajny, używając specyfikatora constexpr gdzie tylko to możliwe, eliminując zbędne kopiowania itp.

## Scenariusz A

Pierwszy scenariusz testowy składał się z dwóch części, wykonywanych na tym samym kontenerze, std::vector posiadającym 100000 elementów.

Pierwsza część zakładała wypełnienie kontenera z pomocą funkcji std::generate, używając do tego wyrażenia lambda (w przypadku C++03 funktora).
Lambda zwracała różnicę dwóch obliczonych wyrażeń. Pierwsze wyrażenie składało się z prostych operacji dodawania oraz mnożenia. Drugie wyrażenie to obliczenie trzydziestego wyrazu Ciągu Fibonacciego metodą iteracyjną.

Druga część polegała na odejmowaniu rosnącej liczby całkowitej od kolejnych elementów kontenera, zapisanej w zmiennej globalnej, zaczynając od 0. Zrealizowano to za pomocą funkcji std::for_each, która pozwala na wywołanie wyrażenia lambda bądź funktora dla każdego elementu.

## Scenariusz B

Drugi scenariusz także składał się z dwóch części. Kontenerem na którym wykonywano operacje również był std::vector o pojemności 100000.

Pierwsza część odpowiadała za wypełnienie kontenera używając do tego wyrażenia lambda (bądź funktora). Użyto funkcji std::generate.

Zadana była tablica składająca się z 20 elementów o wartościach od 1 do 20. Polecenie było takie, aby na każdym elemencie można było wykonać dowolną operację, a następnie zsumować wynik wszystkich takich operacji do zadanej wcześniej wartości inicjalizującej. Koniecznym wymaganiem było to, aby ta dowolna operacja przekazana była jako argument, przyjmująca jeden parametr. W tym przypadku zadaną liczbę podnoszono do kwadratu.

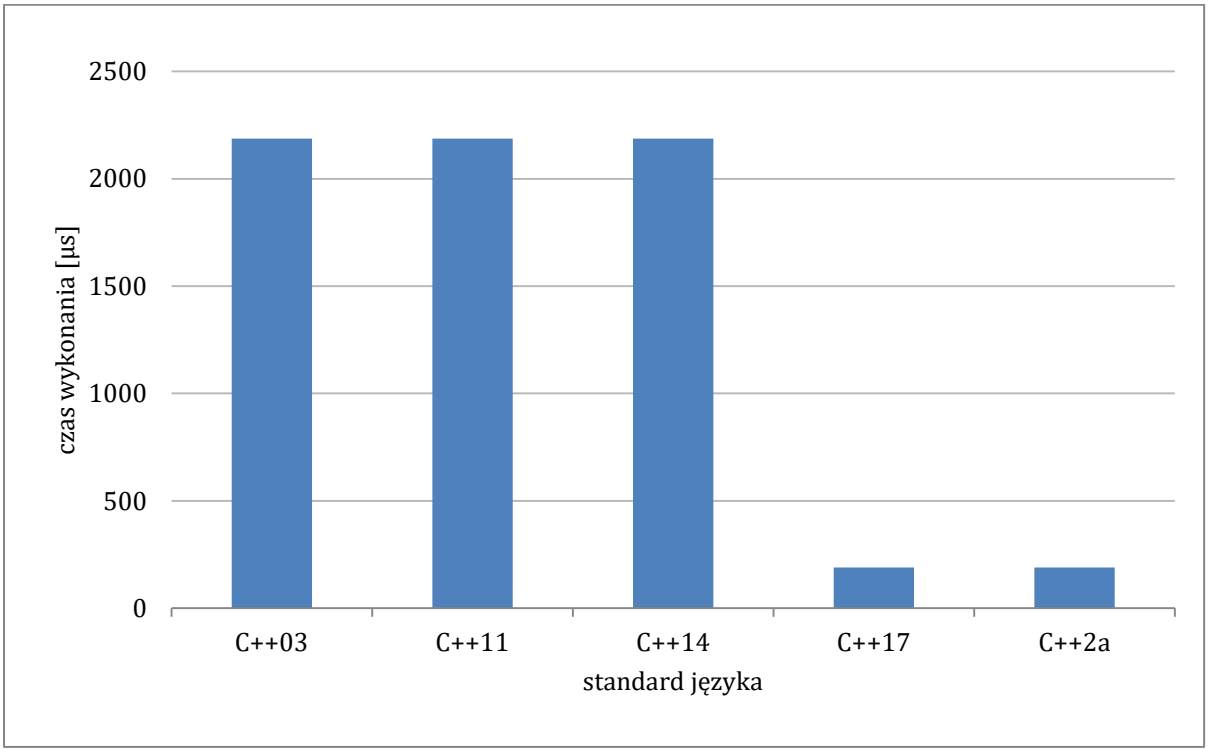
Druga część była identyczna, jak dla scenariusza A.

# 10. Wyniki pomiarów

W celu minimalizacji błędu pomiarowego poprzez wpływ zmiennych środowiskowych, przygotowane fragmenty kodu realizujące scenariusze A oraz B zostały uruchomione sto razy. Do porównań wydajności zostały wzięte pod uwagę najmniejsze wartości czasu, gdyż odpowiadają one przypadkowi najszybszego wykonania.

## Scenariusz A

Wyniki najlepszych pomiarów fragmentów kodu dla poszczególnych standardów podczas realizacji scenariusza A przedstawiono w postaci wykresu na rysunku 1.

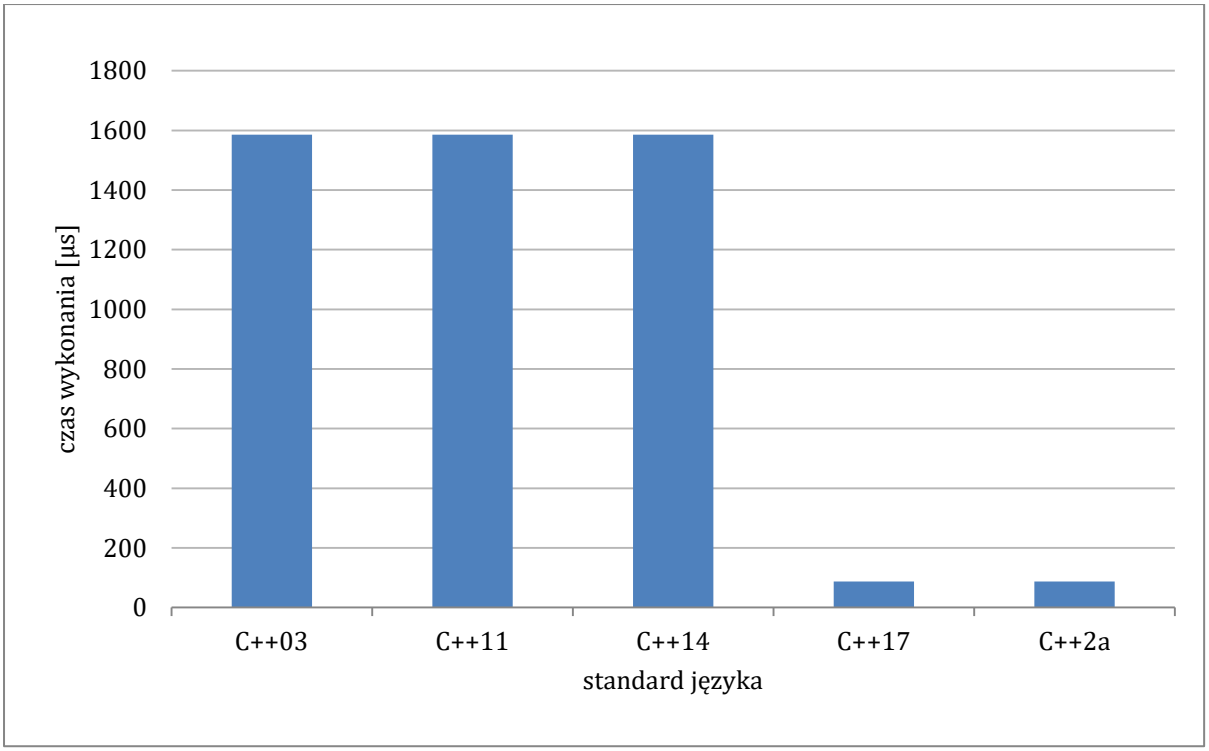



Rys. 1. Czas wykonania scenariusza A dla poszczególnych standardów języka C++

Analizując powyższy wykres można zauważyć, iż kod przygotowany w standardzie C++03, C++11 oraz C++14 wykonywany został w jednakowym czasie (2188 µs). Tak samo jest w przypadku kodu napisanego w standardzie C++17 i C++20 (189 µs), jednak scenariusz został zrealizowany ponad jedenastokrotnie szybciej.

## Scenariusz B

Wyniki najlepszych pomiarów fragmentów kodu dla poszczególnych standardów podczas realizacji scenariusza B przedstawiono w postaci wykresu na rysunku 2.




Rys. 2. Czas wykonania scenariusza B dla poszczególnych standardów języka C++

Analizując powyższy wykres można zauważyć, że kod zrealizowany w standardzie C++03, C++11 i C++14 został zrealizowany w takim samym czasie (1586 µs). Podobnie jest w dla kodu napisanego w standardzie C++17 oraz C++20 (87 µs), jednak scenariusz został wykonany ponad osiemnastokrotnie szybciej.

## Interpretacja wyników

Szczególnie interesujący jest fakt iż w obu scenariuszach C++03 pozwalał na osiągnięcie takiej samej wydajności, co C++11. Oznacza to, że kod maszynowy jest identyczny lub maksymalnie zbliżony w obu przypadkach. Jednak z punktu widzenia programisty użycie starszego standardu jest mało wygodne, gdyż nie ma możliwości użycia funkcji anonimowych, a zamiast tego należy tworzyć funktory o zasięgu globalnym. Można osiągnąć tą samą wydajność poświęcając więcej czasu i tracąc na jakości kodu.

W obu scenariuszach przyczyną gwałtownego wzrostu wydajności w trakcie wykonania kodu jest pomiędzy C++14 a C++17 jest zastosowanie specyfikatora constexpr, czyli uogólnionych wyrażeń stałych. Wyrażenia lambda spełniły wszystkie wymagania, aby zagwarantować, że wartość przez nie zwracana jest niezmienna podczas kompilacji. Dzięki temu znaczna część obliczeń została wykonana przez kompilator, zamiast w czasie działania programu. Co więcej, trzydziesty wyraz Ciągu Fibonacciego został obliczony w trakcie kompilacji tylko jeden raz, zamiast każdorazowo podczas iteracji w funkcji bibliotecznej, podobnie jak wartość inicjalizująca kontener w drugim scenariuszu. W tym przypadku, pozwoliło to na kilkunastokrotny wzrost wydajności.

# 11. Wnioski

Podsumowując, czas wykonania fragmentów programu mocno wykorzystującego wyrażenia lambda wyraźnie spada jedynie wraz ze zmianą standardu C++14 na C++17. Pomiędzy pozostałymi jednak nie następuje wzrost wydajności, mimo iż autor spodziewał się tego w trakcie formułowania problemu. Ponadto zastąpienie funktorów wykorzystywanych w C++03 na wyrażenia lambda wprowadzone w C++11, bez zmian w kodzie większych niż miejscowe, dotyczące tylko tych wyrażeń nie powoduje wzrostu wydajności. Główną zaletą nowszych standardów w kontekście funkcji anonimowych jest możliwość zapewnienia większej czytelności kodu oraz większych możliwości, jeśli chodzi o sposoby implementacji funkcjonalności programów.

Zakładane zadania zostały zrealizowane, a samo pisanie pracy było pouczającym doświadczeniem. W tym czasie autor mógł zapoznać się z wieloma technicznymi dokumentami, z którymi większość programistów na co dzień nie ma do czynienia, ponieważ korzystają oni jedynie z łatwo przyswajalnych źródeł wiedzy, co nie pozwala niestety jedynie na powierzchowne zbadanie tematu.

# 12. Bibliografia


[1] URL: https://www.iso.org/standard/38110.html
Data dostępu: 22.02.2020

[2] URL: https://isocpp.org/std/the-committee
Data dostępu: 22.02.2020

[3] URL: https://isocpp.org/wiki/faq/cpp11#cpp11-approach
Data dostępu: 22.02.2020

[4] URL: https://www.iso.org/standard/50372.html
Data dostępu: 22.02.2020

[5] URL: https://timsong-cpp.github.io/cppwp/n3337/
Data dostępu: 22.02.2020

[6] URL: https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#1
Data dostępu: 22.02.2020

[7] URL: https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#2
Data dostępu: 22.02.2020

[8] URL: https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#3
Data dostępu: 22.02.2020

[9] URL: https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#19
Data dostępu: 22.02.2020

[10] URL: https://timsong-cpp.github.io/cppwp/n3337/expr.prim.lambda#5
Data dostępu: 22.02.2020

[11] URL: https://en.cppreference.com/w/cpp/language/inline
Data dostępu: 22.02.2020

[12] URL: https://en.cppreference.com/w/cpp/language/member_functions#const-.2C_volatile-.2C_and_ref-qualified_member_functions
Data dostępu: 22.02.2020

[13] URL: https://en.cppreference.com/w/cpp/language/ub
Data dostępu: 22.02.2020

[14] URL: https://isocpp.org/files/papers/N3797.pdf § 17.6.3.2, Tabela 20, 22
Data dostępu: 22.02.2020

[15] URL: https://en.wikipedia.org/wiki/Immediately_invoked_function_expression
Data dostępu: 22.02.2020

[16] URL: https://isocpp.org/files/papers/N3797.pdf § 8.4.5.1
Data dostępu: 22.02.2020
[17] URL: https://www.iso.org/standard/64029.html
Data dostępu: 22.02.2020
[18] URL: https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda#5
Data dostępu: 22.02.2020
[19] URL: https://en.wikipedia.org/wiki/Segmentation_fault
Data dostępu: 22.02.2020
[20] URL: https://timsong-cpp.github.io/cppwp/n4140/expr.prim.lambda#6
Data dostępu: 22.02.2020
[21] URL: https://www.iso.org/standard/68564.html
Data dostępu: 22.02.2020
[22] URL: https://en.cppreference.com/w/cpp/language/constexpr
Data dostępu: 22.02.2020
[23] URL: https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda#closure-6
Data dostępu: 22.02.2020
[24] URL: https://timsong-cpp.github.io/cppwp/n4659/expr.prim.lambda#capture-1
Data dostępu: 22.02.2020
[25] URL: https://docs.microsoft.com/en-gb/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019
Data dostępu: 22.02.2020
[26] URL: https://www.iso.org/standard/79358.html
Data dostępu: 22.02.2020
[27] URL: https://isocpp.org/std/status
Data dostępu: 22.02.2020
[28] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0409r2.html
Data dostępu: 22.02.2020
[29] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0428r2.pdf
Data dostępu: 22.02.2020
[30] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0780r2.html
Data dostępu: 22.02.2020
[31] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1091r3.html
Data dostępu: 22.02.2020

[32] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0624r2.pdf
Data dostępu: 22.02.2020
[33] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0315r4.pdf
Data dostępu: 22.02.2020
[34] URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0588r1.html
Data dostępu: 22.02.2020
[35] URL: https://cppinsights.io/about.html
Data dostępu: 22.02.2020
[36] URL: https://www.boost.org/doc/libs/1_49_0/doc/html/chrono.html
Data dostępu: 22.02.2020