C++11…17…20 i nowsze - rdzeń języka i ulepszenie funkcjonalności

Wykład 10

C++11 motywacja

- C++11, zwany inaczej C++0x (dawniej) został wprowadzony 2011 roku
- Następca standardu C++03
- Wprowadza wiele udogodnień i poprawek
  - W samym języku
  - Jak i w bibliotekach
- Kompilatory powoli dostosowują się do niego
  - Generalnie wszystkie nowe dystrybucję systemów operacyjnych zawierają już kompilator zgodny z tym standardem
- 5 grudnia 2014 roku został wydany standard C++14 (C++1y)
- 15 grudnia 2017 roku został wydany standard C++17 (C++1z)
- W drodze jest standard C++20 (C++2a)

C++11 i kolejne motywacja

- Utrzymywać stabilność i kompatybilność z C++98 i być może z C,
- Preferować wprowadzanie nowych możliwości przez rozszerzenie biblioteki standardowej zamiast rozszerzenia rdzenia języka,
- Preferować zmiany mogące rozwijać techniki programistyczne,
- Ulepszać C++ tak, aby ułatwić projektowanie systemów i bibliotek zamiast wprowadzać nowe możliwości, które mogłyby być przydatne tylko w szczególnych zastosowaniach,
- Zwiększać bezpieczeństwo typów poprzez wprowadzenie bezpieczniejszych zamienników aktualnych, mniej bezpiecznych technik,
- Zwiększać wydajność i zdolność bezpośredniej współpracy ze sprzętem,
- Dostarczyć odpowiednich rozwiązań dla rzeczywistych problemów praktycznych,
- Uczynić C++ łatwiejszym do nauczania i uczenia się bez usuwania narzędzi potrzebnych ekspertom.

Dokumentacja / literatura

- http://en.cppreference.com
- http://www.cplusplus.com
- https://isocpp.org/
- http://www.stroustrup.com/Tour.html
- https://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/
- http://thispointer.com/c11-tutorial/
- Książka po polsku
  - http://helion.pl/ksiazki/jezyk-c-kompendium-wiedzy-wydanie-iv-bjarne-stroustrup,jcppkw.htm
- Online kompilatory
  - https://godbolt.org/
  - http://cpp.sh/
  - https://repl.it/

Najważniejsze nowości z „core"

- Nowe słowa kluczowe
- Nowe typy fundamentalne
- Nowa pętla for
- Silne typy wyliczeniowe
- Referencja do r-wartości
- Tradycyjne Struktury Danych
- Listy inicjalizacyjne
- Usprawnienie konstruowania obiektów
- Nowa składnia funkcji

Najważniejsze nowości z „core" …

- Wyrażenia lambda
- Statyczne asercje
- Operator `sizeof...()`
- Jawne operatory konwertujące
- Usunięcie problemu trójkątnego nawiasu
- Szablony ze zmienną listą parametrów
- Nowe literały łańcuchowe
- Raw `string literal`
- Literały definiowane przez użytkownika
- Identyfikatory `override` and `final`
- …

Nowe słowa kluczowe

- **`decltype`**
  - Sprawdza typ w argumencie i na jego podstawie może stworzyć nową zmienną
  - Dwie formy
    - `decltype(x)`
    - `decltype((x))` – l-wartość na ogół `const&`
  - Przykład 1
- **`auto`**
  - Pozwala na określenie typu na podstawie wyrażenia inicjalizującego
  - Bardzo przydatne w szablonach
  - `auto a = 1 + 2;`
  - Przykład 2

Nowe słowa kluczowe

- **override**
  - W deklaracji funkcji składowej gwarantuje że funkcja jest wirtualna i że przeładowuje istniejącą funkcję z klasy bazowej
  - ```
    struct A { virtual void foo(); void bar(); };
    struct B : A {
    void foo() const override; // Error: signature mismatch
    void foo() override; // OK: B::foo overrides A::foo
    void bar() override; // Error: A::bar is not virtual
    };
    ```
- **final**
  - Zapewnia że funkcja jest wirtualna oraz to iż nie może być przeładowywana w klasach pochodnych
  - Zapewnia że klasa będzie finalna (nie można po niej dziedziczyć)
  - ```
    struct A {
    virtual void foo() final; // A::foo is final
    void bar() final; // Error: non-virtual function cannot be final
    };
    struct B final : A // struct B is final {
    void foo(); // Error: foo cannot be overridden as it's final in
    A };
    struct C : B // Error: B is final { };
    ```

Nowe słowa kluczowe

- **`alignof`**
  - Nowy operator
  - Zwraca ułożenie danych w bajtach
  - Może być wywoływany dla normalnego typu, tablicy czy też referencji
- **`alignas`**
  - Może być zastosowany do deklaracji zmiennej ale nie pola bitowego będącego składnikiem klasy
  - Może też być zastosowany do klas, unii i enum
    - `alignas(128) char cacheline[128];`
    - `struct alignas(16) Test{};`
- Przykład 3 i 4

Nowe słowa kluczowe

- **`nullptr`**
  - Jest „czystą r-wartością" – nie ma nazwy i może być przenoszone
  - Posiada typ **`std::nullptr_t`**
  - Odpowiednik makra **`NULL`**
  - Możliwa jest niejawna konwersja do dowolnego wskaźnika
    - Zwykłego
    - Do składnika klasy/struktury
- Przykład 5

Nowe słowa kluczowe

- **`constexpr`**
  - Deklaruje że można dane wyrażenie lub wartość funkcji obliczyć w czasie kompilacji
  - Użycie w stosunku do deklaracji obiektów wymusza deklarację typu **`const`**
  - Wartość musi spełniać następujące reguły
    - Musi być literałem
    - Musi mieć od razu wartość (inicjalizacja)
    - Jeżeli do konstrukcji wykorzystywane jest wyrażenie to też musi być **`constexpr`**
  - Funkcja musi spełniać następujące reguły
    - Nie może być **`virtual`**
    - Musi zwracać literał
    - Każdy z parametrów musi być literałem
- ```
  constexpr int Get() {return 5;}
  constexpr int a = Get();
  ```
- Przykład 6
  ```

Nowe słowa kluczowe

- `static_assert(bool_constexpr, message)`
  - Może pojawić się w obszarze bloku
  - Może również być wewnątrz klasy
  - Jeżeli `bool_constexpr` zwraca `true` to deklaracja nie ma znaczenia
  - Jeżeli `bool_constexpr` zwraca `false` to deklaracja wywołuje błąd kompilacji razem z wypisaniem informacji zawartej w `message`
- `constexpr int GetInt(int i) { return i - 1; }`
- Przykładowy kod
  - ```
    constexpr int GetInt(int i) { return i - 1; }
    int main()
    { static_assert(GetInt(2) > 0, "Error");
      static_assert(GetInt(1) > 0, "Error"); }
    ```
  - ```
    main.cpp:4:3: error: static_assert failed "Error"
      static_assert(GetInt(1) > 0, "Error");
      ^ ~~~~~~~~~~~~~~~
    1 error generated.
    ```
- Przykład 7

Nowe słowa kluczowe

- `noexcept`
- `noexcept(wyrażenie stałe /--> bool/)`
  - Specyfikuje że dana funkcja nie będzie wyrzucać wyjątków, jest poprawioną wersją `throw()`
  - Nie będzie wywoływać `std::unexpected` i nie musi odwikływać stosu, przez co implementacja może być zrobiona bez narzutu dla w czasie wykonania programu
  - Deklaracje
  - ```
    void f() noexcept;
    void f(); // error, incompatible exception specifications
    void g() noexcept(false);
    void g(); // ok
    ```
  - Jeśli pojawi się wyjątek oczywiście wywoła `std::terminate`
- `noexcept()`
  - Zwraca wartość typu `bool`
  - Operator, który w czasie kompilacji sprawdza czy funkcja deklaruje że nie będzie niczego rzucać
  - Przydatny przy szablonach razem z powyższym specyfikatorem do określenia czy dla danego typu funkcja będzie coś rzucać czy nie
  - Może oczywiście też być użyty w `static_assert`
- Przykład 8
    ```

Nowe słowa kluczowe

- **`thread_local`**
  - Zmienna przynależna do wątku
  - Dla każdego wątku inna kopia
  - Czas życia zgodny z czasem życia danego wątku
  - Może być łączona ze **`static`** oraz **`extern`**
  - Dzięki temu nie musimy tworzyć zabezpieczeń dla danej zmienne w czasie działania wątków
    - Nie będzie występować np. Race Condition
  - Bardzo wygodne jeśli potrzebujemy przechowywać dane dla każdego wątku z osobna
- Przykład 9

Nowe znaczenie słów kluczowych

- **`default`**
  - Wymusza stworzenie domyślnego konstruktora lub operatora=
    - `class_name() = default;`
    - `class_name & class_name :: operator= ( class_name && ) = default;`
    - Powstaje wtedy domyślna odpowiednia funkcja nawet jeśli automatycznie nie byłaby generowana
- **`delete`**
  - Wymusza brak domyślnego konstruktora lub operatora=
    - `class_name() = delete;`
    - `class_name & class_name :: operator= ( class_name && ) = delete;`
- Działają z wszystkich generowanymi operatorami
- Przykład 10

Typedef i using

- **Typedef**
  - Nie ma różnicy tak jak było do tej pory nowa nazwa dla danego typu
  - `typedef std::vector<int> vec_int;`
- **Using** - aliasy
  - `using vec_int  = std::vector<int>;`
  - Różnica w stosunku do szablonów
  - Można wykorzystać nie do końca specyfikując typ
    - ```
      template <typename T>
      using vec_mem = std::vector<T, my_alloc<T>>;
      ```
  - Nie można stosować do specyfikatorów typów (np. `unsigned`)
    - `using Char = char; using Uchar = unsigned Char;`
      ```

Typy wartości

- lvalue – „left value"
  - Jest nazwaną wartością, która nie może być przenaszalna
    - Jest bezpośrednią wartością lub zwracaną przez funkcję w postaci stałej referencji
    - `std::cout, std::cout << 1, ++a, static_cast<int&>(x)`
  - Własności
    - Takie jak glvalue (lvalue or xvalue)
    - Można pobrać adres
    - Może stać po lewej stronie operatora=
    - Może być użyta do inicjalizacji referencji
- rvalue (do C++11) prvalue (od C++11) „pure rvalue"
  - Jest nienazwaną wartością i jest przenaszalna
    - Przykłady: `42, true, nullptr, a++, a+b, a==b, &a, static_cast<int>(x)`, wyrażenia lambda
  - Własności
    - Takie same rvalue (prvalue or xvalue)
    - Nie może być polimorficze
    - Nie może być stałe

Typy wartości

- xvalue – „expiring value" (od C++11)
  - Jest nazwaną wartością która może być przenaszalna
    - Jest zwracaną przez funkcję w postaci referencji
    - `std::move(x), static_cast<char&&>(x), a[n]`
  - Własności
    - Takie jak gvalue i rvalue
    - Może być stałe i polimorficzne
- gvalue – „generalized lvalue"
  - Jest nazwaną wartością i może ale nie musi być przenaszalna, czyli może być albo lvalue albo xvalue
- rvalue – „right value"
  - Jest przenaszalna i nie musi być nazwana, czyli może być prvalue albo xvalue
  - Własności
    - Nie można pobrać adresu: `&a++`
    - Nie może stać po lewej stronie operatora =

Referencje do r-wartości

- W C++ obiekty tymczasowe (określane jako r-wartości), mogą być przekazywane do funkcji, ale tylko jako referencje do stałej
  - Z punktu widzenia takiej funkcji nie jest możliwe rozróżnienie pomiędzy aktualną r-wartością a zwykłym obiektem przekazanym jako `const &`
- C++11 wprowadza nowy typ referencyjny, zwany referencją do r-wartości,
  - `typename &&`
  - Może być akceptowany jako nie-stała wartość, co pozwala obiektom na ich modyfikację.
  - Taka zmiana umożliwia pewnym obiektom na stworzenie semantyki przenoszenia.
- Przykład 11

Referencje do r-wartości

- Bardzo dobrym przykładem jest `std::vector`
  - Reprezentujący zwykłą tablicę i jej rozmiar
  - Jeśli obiekt tymczasowy typu `vector` jest tworzony lub zwracany przez funkcję, to może być przechowany tylko przez stworzenie nowego obiektu `vector` wraz z kopiami wszystkich r-wartości.
- Dlatego wymyślono przenoszenie
  - Przenoszone są wskaźniki a niszczenie pustego obiektu jest szybkie
- Dla bezpieczeństwa nazwana zmienna nigdy nie będzie traktowana jak r-wartość
  - Dlatego wprowadzona przenaszalne konstruktory
  - Przenaszalny `operator=`
  - Oraz szablon `std::move`

Nowe konstruktory

- Przenaszalny konstruktor
  - `T::T(const T&&)` lub `T::T(T&&)`
  - Reguły tworzenia domyślnego konstruktora przenaszalnego
    - Brak implementacji konstruktora kopiującego
    - Brak implementacji operatora przypisania (zwykłego i przenaszalnego)
    - Brak destruktora (domyślny)
  - Powstanie wtedy konstruktor o sygnaturze
    - `T::T(T&&)`
    - Będący publicznym `inline` i nie `explicit` składnikiem klasy `T`
  - Jeżeli jest „trywialny" wykorzystuje do przenoszenia `std::memmove`
  - Trywialny znaczy
    - Generowany automatycznie
    - T nie posiada funkcji wirtualnych i wirtualnych klas bazowych
    - Trywialny jest przenaszalny konstruktor dla klas bazowych oraz składników
- Przykład 12

Konstruktory delegowane

- Konstruktor może wywoływać inny konstruktor w danej klasie
- Bardzo wygodne
- Redukuje potrzebę istnienia metody `init()` i powtarzanie wystąpień na liście inicjalizacyjnej
- Wywołanie innego konstruktora znajduje się na liście inicjalizacyjnej
- Przykład 12a

Copy elision

- Kompilator ma obowiązek pominąć kopiowanie (przenoszenie) pod warunkami
  - W inicjalizacji
    - `T x = T(T(T()));`
  - Przy wywołaniu funkcji kiedy po słowie return stoi prvalue i jest zgodność typu zwracanego z deklaracją
    - `T f() { return T{}; }  T x = f();`
- Kompilator może pominąć kopiowanie (przenoszenie) pod warunkami

Nowy operator=

- Jest to funkcja składowa niestatyczna i nieszablonowa o nazwie operator=
  - `class_name & class_name :: operator= ( class_name && )`
  - Funkcja wywoływana jest kiedy pojawia się po lewej stronie =, a po jego prawe stoi rvalue
  - „Kradnie" zasoby obiektu stojącego po prawej stronie
    - np. dla std::string zostawia po prawej stronie obiekt pusty
- Generowana automatycznie w sytuacji kiedy
  - Nie ma konstruktora kopiującego (niedomyślnego)
  - Nie ma konstruktora przenoszalnego (niedomyślnego)
  - Nie ma kopiującego operatora=
  - Nie ma destruktora
  - Generowany jest wtedy publiczny i inline `T& T::operator=(T&&)`
- Jeżeli jest „trywialny" wykorzystuje do przenoszenia `std::memmove`
  - Trywialny znaczy
    - Generowany automatycznie
    - T nie posiada funkcji wirtualnych i wirtualnych klas bazowych
    - Trywialny jest przenaszalny operator= dla klas bazowych oraz składników
- Przykład 13

Inicjalizacja zmiennych w klasie

- Standardowo lista inicjalizacyjna – znany sposób
- Można także bezpośrednio przypisać do zmiennej wartość
  - W przypadku wystąpienia obu sytuacji na raz liczy się to co jest na liście inicjalizacyjnej konstruktora
  - ```
    struct S {
        int m = 5;
        int n = 7;
        int x = m + 1;
    };
    ```
- Przykład 14
    ```

Listy inicjujące

- Inicjalizują obiekt za pomocą listy zawartej w nawiasach {}
  - Bezpośrednio
    - `T object { arg1, arg2, ... };`
    - `T { arg1, arg2, ... };`
    - `new T { arg1, arg2, ... }`
    - `Class { T member { arg1, arg2, ... }; };`
      - Niestatyczny składnik
    - `Class::Class() : member{arg1, arg2, ...} {...}`
      - Konstruktor przy definicji
      - Przykład 17
  - Kopiując listę inicjalizacyjną
    - `T object = {arg1, arg2, ...};`
    - `function( { arg1, arg2, ... } ) ;`
    - `return { arg1, arg2, ... } ;`
    - `object[ { arg1, arg2, ... } ] ;`
    - `object = { arg1, arg2, ... } ;`
    - `U( { arg1, arg2, ... } )`
    - `Class { T member = { arg1, arg2, ... }; };`
  - Nie ma typu dlatego nie można jej zadeklarować
    - Ale działa z `auto` powodując traktowania takiej listy jak `std::initializer_list`
    - `std::initializer_list` – lekki typu do opakowania {}
    - Przykład 18 (z szablonami)

Tradycyjne Struktury Danych (POD)

- Restrykcje w standardzie c++03 były nieco za silne i zostały zmienione na następujące
  - Klasa/struktura jest uważana za TSD, jeśli jest trywialna, standardowo ułożona i nie posiada żadnych niestatycznych składowych niebędących TSD-ami
    - Posiada trywialny konstruktor domyślny
    - Posiada trywialny konstruktor kopiujący
    - Posiada trywialny operator przypisania
    - Posiada trywialny destruktor, który nie może być wirtualny.
  - Klasa jest standardowo ułożona jeżeli
    - Posiada tylko niestatyczne pola, które są standardowo ułożone,
    - Posiada ten sam poziom dostępu (private, protected, public) dla wszystkich niestatycznych składowych,
    - Nie posiada wirtualnych metod,
    - Nie posiada wirtualnych klas bazowych,
    - Posiada tylko standardowo ułożone klasy bazowe.

Pela for w zakresie

- Przydatna do iterowania po zakresie
  - `for ( range_declaration : range_expression ) loop_statement`
  - Często wykorzystywana razem z `auto`
  - Preferowana forma
    - `for(auto&& var : sequence)`
- Posiada bardzie intuicyjną formę w stosunku do standardowego for
  - Oczywiście kiedy mam do dyspozycji zakres
- Przykład 16

Nowa składnia deklaracji i definicji funkcji

- Składania deklaracji przejęta w jeżyka C nie jest już dopasowana wystarczająco dobrze no nowych potrzeb języka
  - Szczególnie problem przy szablonach
    - ```
      template< typename LHS, typename RHS>
      Ret // NIEPRAWIDŁOWE!
      AddingFunc(const LHS &lhs, const RHS &rhs) {return lhs + rhs;}
      ```
      - `Ret` – cokolwiek wynikające z sumy lhs + rhs
    - ```
      template< typename LHS, typename RHS>
      decltype(lhs + rhs) // NIEPRAWIDŁOWE!
      AddingFunc(const LHS &lhs, const RHS &rhs) {return lhs + rhs;}
      ```
      - Jest to nielegalne w C++ ponieważ lhs i rhs nie są jeszcze zdefiniowane
- C++11 wprowadza nową składnię deklaracji i definicji funkcji:
  - ```
    template< typename LHS, typename RHS>
    auto AddingFunc(const LHS &lhs, const RHS &rhs)
      -> decltype(lhs + rhs)
    {return lhs + rhs;}
    ```
- C++14 naprawia problem dedukcji typu zwracanego:
  - ```
    template< typename LHS, typename RHS>
    auto AddingFunc(const LHS &lhs, const RHS &rhs)
    {return lhs + rhs;}
    ```
- Dotyczy to oczywiście zwykłych funkcji i funkcji składowych
- Przykład 15

Wyrażenia lambda

- Inaczej nienazwane obiekty funkcjne
  - `[ capture-list ] ( params ) mutable(optional) exception -> ret { body }`
  - `[ capture-list ] ( params ) -> ret { body }`
  - `[ capture-list ] ( params ) { body }`
  - `[ capture-list ] { body }`
- Znaczenie
  - `mutable` – pozwala na modyfikacje obiektów przesłanych przez wartość wewnątrz funkcji
  - `exception` – pozwala na określenie specyfikacji wyjątków dla operatora()
  - `capture-list` – lista przyjmowanych zmienny automatycznych
    - [a,&b] a jest przesyłane przez wartość a b przez referencję.
    - [&] wszystkie automatyczne zmienne przez referencję
    - [=] wszystkie automatyczne zmienne przez wartość
    - [this] - obsługa wskaźnika this, wskazującego na obiekt obsługiwany przez daną metodę, jest specjalna i musi być wyraźnie zaznaczona w funkcji lambda
    - [] – nic nie jest przekazywanie
  - `params` – lista parametrów (jak w funkcji)
  - `ret` – typ zwracany
  - `body` – ciało funkcji
- Przykład 19 i 20

Inne ciekawostki

- **`extern template`**
  - Zmusza kompilator do nietworzenia instancji szablonu w danej jednostce kompilacji
  - Działa pod warunkiem, że gdzieś indziej taka instancja szablonu zostanie stworzona
    - Znacząco może przyspieszyć proces kompilacji
- Aliasy typów i szablony (**`using`**)
  - W stosunku do typów odpowiednik **`typedef`**
  - Alias do szablonu może uprościć operacje na typach z jednej rodziny
  - Przykład 21

Usprawnienie obsługi wyjątków

- `std::exception_ptr`
  - Jest obiektem podobnym do wskaźnika
  - Zarządza wyrzuconym obiektem, który został złapany przez `std::current_exception`
  - Instancja `std::exception_ptr` może być przekazana do innej funkcji a nawet do innego wątku gdzie wyjątek może zostać wyrzucony ponownie i przetworzony
  - Dwie instancje `std::exception_ptr` są sobie równe jeżeli są puste lub pokazują na ten sam obiekt wyjątku
  - Obiekt wyjątku jest dostępny do czasu kiedy chociaż jeden `std::exception_ptr` pokazuje na niego
    - Jak zmyślny wskaźnik
- `std::current_exception()`
  - Jeśli wywołana w bloku `catch` to łapie obecny wyjątek i tworzy `std::exception_ptr`
- `std::rethrow_exception(std::exception_ptr)`
  - Wyrzuca wyjątek przechowywany w `std::exception_ptr`
- Przykład 22

async, future (i promise, package_task() ...)

- Pozwalają na wykonywanie współbieżności zadaniowej
  - Nie wymaga skupiania się na wątkach
    - Problemach synchronizacji, unikaniu race conditions etc.
- **`std::async`**
  - Uruchamia funkcję asynchronicznie (potencjalnie w innym wątku)
  - Zwraca **`std::future`**, który zawierać będzie rezultat
- **`std::future`**
  - Zapewnia mechanizm dostępu do wyniku asynchronicznej operacji
- Przykład 23 i 24

Inne ciekawostki

- Literały łańcuchowe
  - `u8"I'm a UTF-8 string."`
  - `u"This is a UTF-16 string."`
  - `U"This is a UTF-32 string."`
  - `R"(The String Data \ Stuff " )"`
  - `R"delimiter(The String Data \ Stuff " )delimiter"`
- Silnie typowane wyliczenia
  - Każdy typ będzie inny i nie da się ich bezpośrednio porównywać
  - Nie ma niejawnej konwersji do int
  - `enum class Enum2 : {Val1, Val2};`
  - `enum class Enum2 : unsigned int {Val1, Val2};`
  - `Enumeration::Val2 == 101 //Błąd kompilacji`

Inne ciekawostki

- Usunięcie problemu trójkątnego nawiasu <>
  - W C++11 w fazie leksykalnej analizy znak ">" będzie interpretowany jako zamykający nawias trójkątny nawet wtedy, gdy jest natychmiast następowany przez ">" lub "=",
    - Naprawia to błędy takie jak
      - `typedef std::vector<std::vector<int> > Table; // Ok.`
      - `typedef std::vector<std::vector<bool>> Flags; //Błąd! ">>" interpretowane jako przesunięcie bitowe na prawo`
    - Jedynie pozostawia problem (ale dużo rzadziej występujący)
      - `X< 1>2 > x1; // Błąd`
      - `X<(1>2)> x1; // Ok.`
- Jawne operatory przekształcenia
  - operator T() (w szczególności do bool – zmyślnie wskaźniki)
  - Słowo **`explicit`** może być w od c++11 stosowane do tych operatorów, dzięki czemu nie będzie dalszych niejawnych przekształceń