Nowoczesny C++

- Kamil Szatkowski, kamil.szatkowski@nokia.com
- Łukasz Ziobroń, lukasz.ziobron@nokia.com

- 2016-03-30

O autorach

Kamil Szatkowski

- Absolwent PWr, Wydział IZ
- C++ software engineer - Nokia, LTE, CPlane
- Code Reviewer
- Okazyjny trener (Nowoczesny C++, Zarządzanie pamięcią w C++)
- Prelegent (AMPPZ)
- Blogger (netrix.org.pl)

Łukasz Ziobroń

- Absolwent PWr, Wydział EKA
- C++ software engineer - Nokia, LTE, LOM
- Scrum Master
- Okazyjny trener (Nowoczesny C++, Zarządzanie pamięcią w C++)
- Prelegent (code::dive 2015, AMPPZ)
- Blogger (ziobron.net)

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Agenda

- **Wprowadzenie do standardu C++14**
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Wprowadzenie do standardu C++14

Historia standaryzacji C++

- 1998 - pierwszy standard ISO C++
- 2003 - TC1 (“Technical Corrigendum 1”) opublikowane jako (“C++03”). Poprawki błędów (bug fixes) dla C++98
- 2005 - opublikowany “Technical Report 1”
- 2011 - ratyfikowany C++0x -> C++11
- 2013 - pełna wersja draftu C++14
- 2014 - opublikowany C++14 (minor revision)
- 2017? - planowana duża modyfikacja standardu jako C++17

Wprowadzenie do standardu C++14
Nowości

Nowe słowa kluczowe:

- *alignas*
- *alignof*
- *char16_t*
- *char32_t*
- *constexpr*
- *decltype*
- *noexcept*
- *nullptr*
- *static_assert*
- *thread_local*

Elementy przestarzałe (deprecated):

- Funkcje wiążące *bind1st*, *bind2nd*, itp.
- Specyfikacja rzucanych wyjątków przez funkcje
- Klasa *auto_ptr<>*
- Specyfikator *register*

Wprowadzenie do standardu C++14
Wsparcie w kompilatorach

Wsparcie dla C++11

- Wybrane elementy C++11 - gcc4.3, clang2.9
- Pełne wsparcie - gcc4.6, clang3.3
- Flagi kompilacji:
  - -std=c++0x
  - -std=c++11 - od gcc4.7, clang3.3
- Szczegóły:
  - http:://gcc.gnu.org/projects/cxx0x.html
  - http://clang.llvm.org/cxx_status.html

Wsparcie dla C++14

- Podstawowe funkcjonalności - gcc4.9, clang3.3
- Pełne wsparcie – gcc5, clang3.4
- Opcja kompilacji:
  - -std=c++1y
  - -std=c++14
- Szczegóły:
  - http:://gcc.gnu.org/projects/cxx1y.html
  - http://clang.llvm.org/cxx_status.html

Agenda

- Wprowadzenie do standardu C++14
- **Alias ‚using'**
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Alias ‚using'

C++11 wprowadza aliasy na uprzednio znane typy (podobne do typedef)

```
using flags = std::ios_base::fmtflags; // equal to typedef std::ios_base::fmtflags flags;
flags fl = std::ios_base::dec;

using SocketContainer = std::vector<std::shared_ptr<Socket>>;
typedef std::vector<std::shared_ptr<Socket>> SocketContainer;
std::vector<std::shared_ptr<Socket>> typedef SocketContainer;
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- **Uniwersalny pusty wskaźnik *nullptr***
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Uniwersalny pusty wskaźnik *nullptr*

Nowe słowo kluczowe - *nullptr:*

- wartość dla wskaźników, które na nic nie wskazują,
- bardziej czytelny i bezpieczniejszy odpowiednik stałej NULL/0,
- posiada zdefiniowany przez standard typ - std::nullptr_t,
- rozwiązuje problem z przeciążeniem funkcji przyjmujących jako argument wskaźnik lub typ całkowity.

Uniwersalny pusty wskaźnik *nullptr*
Przykłady

```
int* p1 = nullptr;
int* p2 = NULL;
int* p3 = 0;

p2 == p1; // true
p3 == p1; // true

int* p {}; // p is set to nullptr


void foo(int);

foo(0); // calls foo(int)
foo(NULL); // calls foo(int)
foo(nullptr); // compile-time error

void bar(int);
void bar(void*);
void bar(nullptr_t);

bar(0); // calls bar(int)
bar(NULL); // calls bar(int) if NULL is 0, ambigous if NULL is 0L
bar(nullptr); // calls bar(void*) or bar(nullptr_t) if provided
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- **Słowo kluczowe *auto***
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Słowo kluczowe *auto*
Deklaracje typu z *auto*

Deklaracje zmiennych z użyciem słowa kluczowego auto umożliwiają automatyczną dedukcję typu zmiennej przez kompilator.

auto jest słowem kluczowym, które w C++11 otrzymało nowe znaczenie.

W poprzednich standardach oznaczało zmienną automatyczną (tworzoną na stosie) - nigdy nie było używane.

Definiując zmienną z użyciem auto można dodawać modyfikatory const, volatile oraz stosować referencje lub wskaźniki.

Używając auto można wygodnie iterować po kontenerach dedukcję typu iteratora zostawiając kompilatorowi.

Aby uzyskać w trakcie iteracji const_iterator należy użyć nowych metod z interfejsu kontenerów standardowych cbegin() i cend().

Słowo kluczowe *auto*
Przykłady

```
auto i = 42;            // i : int
const auto *ptr_i = &i; // ptr_i : const int*

double f();
auto r1 = f();          // r1 : double
const auto& r2 = f();   // r2: const double&

std::set<std::string> someStringSet;
auto it = someStringSet.begin();               // it : std::set<std::string>::iterator
const auto& ref_someStringSet = someStringSet; // ref_someStringSet : const std::set<std::string>&

void do_something(int& x);
void print(const int& x);

std::vector<int> vec = { 1, 2, 3, 4, 5 };

for(auto it = vec.begin(); it != vec.end(); ++it)
{
    do_something(*it);      // it : vector<int>::iterator
}

for(const auto& item : vec) // ok – range-based for
{
    print(item);            // item : const int &
}
```

Słowo kluczowe *auto*
Przykłady

```
const vector<int> values;
auto v1 = values;  // v1 : vector<int>
auto& v2 = values; // v2 : const vector<int>&

volatile long clock = 0L;
auto c = clock;    // c : long

Gadget items[10];
auto g1 = items;   // g1 : Gadget*
auto& g2 = items;  // g2 : Gadget(&)[10] - reference to an array

int func(double) { return 10; }
auto f1 = func;    // f1 : int(*)(double)
auto& f2 = func;   // f2: int(&)(double)
```

Słowo kluczowe *decltype*
Deklaracje typu z *decltype*

Słowo kluczowe *decltype* umożliwia kompilatorowi określenie zadeklarowanego typu dla podanego jako argument obiektu lub wyrażenia.

```
std::map<std::string, float> coll;

decltype(coll) coll2;           // coll2 has type of coll
decltype(coll)::value_type val; // val has type float
```

Nowa składnia deklaracji funkcji

Deklaracje funkcji z typem zwracanym ->

Nowa alternatywna składnia deklaracji funkcji pozwala deklarować typ zwracany po liście parametrów funkcji.

Pozwala to na specyfikację zwracanego typu wewnątrz funkcji oraz z użyciem argumentów funkcji.

W połączeniu z decltype umożliwia specyfikację typu na podstawie wyrażenia wykorzystującego argumenty funkcji.

```
int sum(int a, int b);
auto sum(int a, int b) -> int;


template <typename T1, typename T2>
auto add(T1 a, T2 b) -> decltype(a + b)
{
    return a + b;
}
```

Automatyczna dedukcja typu zwracanego z funkcji (C++14)
Dedukcja z *auto*

W C++14 typ zwracany z funkcji może być automatycznie dedukowany z implementacji funkcji. Mechanizm dedukcji jest taki sam jak mechanizm automatycznej dedukcji typów zmiennych.

Jeśli w funkcji występuje wiele instrukcji return muszą one wszystkie zwracać wartości tego samego typu.

Rekurencja dla funkcji z auto jest możliwa, o ile rekurencyjne wywołanie następuje po przynajmniej jednym wywołaniu *return* zwracającego wartość nierekurencyjną.

Automatyczna dedukcja typu zwracanego z funkcji (C++14)
Przykłady

```
auto multiply(int x, int y)
{
    return x * y;
}

auto get_name(int id)
{
    if (id == 1)
        return "Gadget"s;
    else if (id == 2)
        return "SuperGadget"s;
    return string("Unknown");
}

auto factorial(int n)
{
    if (n == 1)
        return 1;
    return factorial(n-1) * n;
}
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- **Lista inicjalizacyjna**
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Jednolita inicjalizacja zmiennych
Wykorzystanie klamer {} do inicjalizacji zmiennych

W C++11 wprowadzono możliwość inicjalizacji zmiennych przy pomocy klamer {}.

Pozwala to uniknąć wielu problemów znanych z C++98 takich jak:

- most vexing parse,
- brak możliwości inicjalizacji kontenerów z wykorzystaniem list wartości,
- różne sposoby inicjalizacji zmiennych typów prostych, złożonych, struktur oraz tablic.

Dotychczasowy sposób wykorzystujący składnię C++98 jest poprawny z wyjątkiem niejawnej konwersji zawężającej typ.

Jednolita inicjalizacja zmiennych

Przykłady

```
int i;       // undefined value

int va(5);   // c++98: "direct initialization", v = 5
int vb = 10; // c++98: "copy initialization", v = 10
int vc();    // c++98: "function declaration", common error named "most-vexing-parse", compiles normally, but
             // generally this behaviour is not expected

int vd{5};   // c++11: brace initialzation

int values[] = { 1, 2, 3, 4 };   // c++98: brace initialization

struct P { int a, b; };
P p = { 20, 40 };                // c++98: brace initialization

std::complex<float> ca(12.0f, 54.0f);  // c++98: initialization of classes, using constructor
std::complex<float> cb{12.0f, 54.0f};  // c++11: brace initialization, using the same constructor as above

std::vector<std::string> colors;       // c++98: no brace initialization like with simple arrays/structs
colors.push_back("yellow");
colors.push_back("blue");

std::vector<std::string> names = { "John", "Mary" };    // c++11: brace initialization with std::initializer_list
std::vector<std::string> names{ "John", "Mary" };       // c++11: brace initialization with std::initializer_list

int array[] = { 1, 2, 5.5 };     // C++98 - OK, C++11: error - implicit type narrowing
```

Inicjalizacja zmiennych niestatycznych w klasie
brace-or-equal initializer

W C++98 zmienne klas mogły być inicjalizowane wyłącznie na liście inicjalizacyjnej konstruktora lub w jego w ciele. Wyjątkiem były stałe, całkowitoliczbowe zmienne statyczne.

Od C++11 możliwa jest inicjalizacja zmiennych klasy również w jej ciele.

Inicjalizacja taka określa wartości domyślne pól klasy. Mogą być one dalej nadpisane na liście inicjalizacyjnej lub w ciele konstruktora.

Inicjalizacja zmiennych niestatycznych w klasie
Przykład

```
class Foo
{
public:
   Foo()
   {}

   Foo(std::string a) :
      m_a(a)
   {}

   void print()
   {
      std::cout << m_a << std::endl;
   }

private:
   std::string m_a = "Fooooo";          // C++98: error, C++11: OK
   static const unsigned VALUE = 20u;   // C++98: OK, C++11: OK
};

Foo().print();             // Fooooo
Foo("Baar").print();       // Baar
```

Inicjalizacja z wykorzystaniem listy inicjalizacyjnej
std::initializer_list

W C++98 inicjalizacja z wykorzystaniem listy inicjalizacyjnej jest możliwa wyłącznie w przypadku tablic oraz struktur typu POD (Pure Old Data).

Od C++11 wprowadza tę składnię również dla obiektów klas z wykorzystaniem specjalnego szablonu klasy - std::initializer_list.

std::initializer_list wykorzystuje semantykę kopiowania, tj. raz umieszczona wartość nie może być z takiej listy przeniesiona gdzieś indziej (np. trzymany std::unique_ptr).

std::initializer_list posiada kilka funkcji pomocniczych: size(), begin()/end().

Konstruktory wykorzystujące std::initializer_list mają wyższy priorytet niż inne.

Inicjalizacja z wykorzystaniem semantyki listy
Przykład

```
template<class Type>
class Bar
{
public:
   Bar(std::initializer_list<Type> values)
   {
      for(auto a : values)             // only example, can be much better
      {
         m_values.push_back(value);
      }
   }

   Bar(Type a, Type b} :
      m_values{a, b}
   {}

private:
   std::vector<Type> m_values;
};


Bar<int> b = { 1, 2 };                                    // OK, first constructor is used
Bar<int> b = { 1, 2, 5, 51 };                             // OK, first constructor is used
Bar<std::unique_ptr<int>> c = { new int{1}, new int{2} }; // error - std::unique_ptr is non-copyable
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- **Słowa kluczowe *default* i *delete***
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Słowa kluczowe *default* i *delete*

Domyślne funkcje składowe - default

Deklaracja *default* - wymusza na kompilatorze generację domyślnej implementacji dla deklaracji specyfikowanej przez użytkownika (np. generacja domyślnego konstruktora w przypadku, gdy istnieją inne konstruktory przyjmujące parametry).

Jako default oznaczyć można tylko specjalne funkcje składowe klas: konstruktor domyślny, destruktor, konstruktor kopiujący, kopiujący operator=, konstruktor przenoszący (C++11), przenoszący operator= (C++11)

```
class Gadget
{
public:
    Gadget(const Gadget&); // copy constructor will prevent
                           // generating implicitly declared
                           // default ctor and move operations
    Gadget() = default;
    Gadget(Gadget&&) noexcept = default;
};
```

Słowa kluczowe *default* i *delete*
Usunięte funkcje składowe - *delete*

Deklaracja delete - usuwa wskazaną funkcję lub funkcję składową z interfejsu klasy. Nie jest generowany kod takiej funkcji, a wywołanie jej, pobranie adresu lub użycie w wyrażeniu z sizeof jest błędem kompilacji.

```
class NoCopyable
{
protected:
    NoCopyable() = default;

public:
    NoCopyable(const NoCopyable&) = delete;
    NoCopyable& operator=(const NoCopyable&) = delete;
};

class NoMoveable
{
    NoMoveable(NoMoveable&&) = delete;
    NoMoveable& operator=(NoMoveable&&) = delete;
};
```

Słowa kluczowe *default* i *delete*
Usunięte funkcje składowe - *delete*

Usunięcie funkcji umożliwia uniknięcie niejawnej konwersji argumentów wywołania funkcji.

```
void integral_only(int a)
{
    cout << "integral_only: " << a << endl;
}

void integral_only(double d) = delete;

// ...

integral_only(10);  // OK

short s = 3;
integral_only(s);   // OK - implicit conversion to short

integral_only(3.0); // error - use of deleted function
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- **Słowa kluczowe *override* i *final***
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Słowa kluczowe *override* i *final*

Wymuszanie przesłaniania przy pomocy *override*

Deklaracja *override* wymusza na kompilatorze sprawdzenie czy dana funkcja przesłania funkcję wirtualną w klasie bazowej.

```
struct A
{
   virtual void foo() = 0;
   void dd() {}
};

struct B : A
{
void foo() override {}     // OK, method overrides in base class
void bar() override {}     // error, there is no virtual method in struct A
void dd() override {}      // error, only virtual methods can be overridden
}
```

Słowa kluczowe *override* i *final*
Blokowanie dziedziczenia przy pomocy *final*

Deklaracja *final* użyta przy nazwie klasy nie pozwala na stworzenie nowej klasy, która po niej dziedziczy.

```
struct A final
{
};

struct B : A        // error, cannot derive from class marked as final
{
};
```

Słowa kluczowe *override* i *final*
Blokowanie przesłaniania przy pomocy *final*

Deklaracja *final* użyta przy nazwie deklaracji funkcji wirtualnej uniemożliwia jej przesłonięcie w klasie pochodnej.

```
struct A
{
   virtual void foo() const final
   {}

   void bar() const final              // error, only virtual functions can be marked as final
   {}
};

struct B : A
{
   void foo() const override           // error, cannot override function marked as final
   {}
};
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- **Inteligentne wskaźniki**
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Inteligentne wskaźniki
Mechanizm wyjątków a zasoby

Używanie natywnych wskaźników (*raw pointers*) do zarządzania zasobami może powodować wycieki zasobów. W celu zabezpieczenia przed wyciekiem zasobu możemy użyć konstrukcji try-catch.

Niestety w rezultacie kod staje się mało czytelny i występuje w nim duplikacja zwalniania zasobu.

```
void use_resource()
{
    Resource* rsc = nullptr;
    try
    {
        rsc = new Resource();
        rsc->use(); // Kod, który używa rsc i może rzucić wyjątkiem
        may_throw();
    }
    catch(...) //Przechwytuje wszystkie wyjątki
    {
        delete rsc;
        throw;
    }
    delete rsc;
}
```

Inteligentne wskaźniki
std::unique_ptr<T>

Klasa szablonowa *std::unique_ptr* służy do zapewnienia właściwego usuwania przydzielanego dynamicznie obiektu.

Implementuje RAII - destruktor inteligentnego wskaźnika usuwa wskazywany obiekt. Wskaźnik *unique_ptr* nie może być ani kopiowany ani przypisywany, może być jednakże przenoszony.

Przeniesienie prawa własności odbywa się zgodnie z move semantics w C++11 - wymaga dla referencji do l-value jawnego transferu przy pomocy funkcji *std::move()*.

Inteligentne wskaźniki

std::unique_ptr<T> - przykłady

```
void f()
{
    std::unique_ptr<Gadget> my_gadget {new Gadget()};
    my_gadget->use(); // kod, który może wyrzucać wyjątki
    std::unique_ptr<Gadget> your_gadget = std::move(my_gadget); // explicit move
} // Destruktor klasy unique_ptr wywoła operator delete dla wskaźnika do kontrolowanej instancji


auto ptr = std::make_unique<Gadget>(arg); // C++14 ptr: std::unique_ptr<Gadget>

void sink(std::unique_ptr<Gadget> gdgt)
{
    gdgt->call_method();
    // sink takes ownership - deletes the object pointed by gdgt
}

sink(std::move(ptr)); // explicitly moving into sink

// pointers to derived classes
std::unique_ptr<Gadget> pb = std::make_unique<SuperGadget>();  // SuperGadget derives from Gadget
auto pb = std::unique_ptr<Gadget>{ std::make_unique<SuperGadget>() };
```

Inteligentne wskaźniki
std::shared_ptr<T>

Inteligentne wskaźniki ze zliczaniem odniesień eliminują konieczność kodowania skomplikowanej logiki sterującej czasem życia obiektów współużytkowanych przez pewną liczbę innych obiektów.

std::shared_ptr jest szablonem wskaźnika zliczającego odniesienia do wskazywanych obiektów.

Działanie:

- konstruktor tworzy licznik odniesień i inicjuje go wartością 1,
- konstruktor kopiujący lub operator przypisania inkrementują licznik odniesień,
- destruktor zmniejsza licznik odniesień, jeżeli ma on wartość 0, to usuwa obiekt wywołując domyślnie operator delete.

Inteligentne wskaźniki

std::shared_ptr<T> - przykłady

```
#include <memory>

class Gadget { /* implementacja */ };

std::map<std::string, std::shared_ptr<Gadget>> gadgets; // it wouldn't compile with C++03. Why?

void foo()
{
    std::shared_ptr<Gadget> p1 {new Gadget(1)};  // reference counter = 1
    {
        auto p2 = p1; // copying of shared_ptr (reference counter == 2)

        gadgets.insert(make_pair("mp3", p2)); // copying shared_ptr to a std container (reference counter == 3)

        p2->use();
    }  // destruction of p2 decrements reference counter = 2
}  // destruction of p1 decrements reference counter = 1

gadgets.clear(); // reference counter = 0 - gadget is removed
```

Inteligentne wskaźniki

std::make_shared<T> i std::make_unique<T>

Używanie *std::shared_ptr* eliminuje konieczność stosowanie operatora delete, jednakże nie eliminuje użycia new. Można uniknąć używania operatora new stosując zamiast tego funkcję pomocniczą make_shared(), która pełni rolę fabryki wskaźników shared_ptr. Funkcja przekazuje swoje parametry do konstruktora obiektu kontrolowanego przez inteligentny wskaźnik (perfect forwarding).

Stosowanie funkcji make_shared() jest wydajniejsze niż konstrukcja shared_ptr(new std::string) ponieważ alokowany jest tylko jeden segment pamięci, w którym umieszczany jest wskazywany obiekt oraz blok kontrolny z licznikami odniesień.

Tak samo jest z make_unique(), które zostało wprowadzone w C++14.

```
auto x = std::make_shared<std::string>("hello, world!"); // std::shared_ptr<std::string>
std::cout << *x << std::endl;


auto ptr = make_unique<Gadget>(arg); // C++14
```

Inteligentne wskaźniki
Zastosowanie

Wskaźniki *std::unique_ptr* należy stosować tam, gdzie:

- w zasięgu obarczonym ryzykiem zgłoszenia wyjątku występuje wskaźnik,
- funkcja ma kilka ścieżek wykonania i kilka punktów powrotu,
- istnieje tylko jeden obiekt zarządzający czasem życia alokowanego obiektu,
- ważna jest odporność na wyjątki.

Wskaźniki *std::shared_ptr* można skutecznie stosować tam, gdzie:

- jest wielu użytkowników obiektu, ale nie ma jednego jawnego właściciela,
- trzeba przechowywać wskaźniki w kontenerach biblioteki standardowej,
- trzeba przekazywać wskaźniki do i z bibliotek, a nie ma jawnego wyrażenia transferu własności.

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- **Semantyka przenoszenia**
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Semantyka przenoszenia
Zalety i nowości

Zwiększona wydajność przez rozpoznanie obiektów tymczasowych i możliwość przeniesienia z nich zawartości zamiast robienia kopii (przeważnie głębokich).

Nowa składnia przez wprowadzenie referencji *r-value* (**auto && value**).

Nowe funkcje składowe klasy:

- konstruktor przenoszący **Class(Class && src),**
- przenoszący operator przypisania **Class& operator=(Class && src).**

Nowe funkcje pomocnicze:

- std::move() - wymuszenie użycia konstruktora przenoszącego lub przenoszącego operatora przypisania
- std::forward() - przekazanie referencji r-value dalej.

Semantyka przenoszenia
l-value a *r-value*

Obiekty l-value posiadają nazwę oraz jest możliwość pobrania ich adresu. Operacja przeniesienia stanu może być niebezpieczna, ponieważ istnieje możliwość dalszego korzystania z takiego obiektu.

Obiekty r-value zwykle nie posiadają nazwy, nie można pobrać ich adresu, można bezpiecznie przenosić ich stan, pozostawiając go poprawnym dla destrukcji.

Referencje l-value - można nich dowiązywać obiekty l-values, referencje r-value oraz wyjątkowo do referencji const l-value również obiekty r-value. Nie można natomiast dowiązać wartości tymczasowych non-const.

Referencje r-value - można do nich dowiązywać obiekty r-value. Nie można natomiast dowiązać obiektów i referencji l-value.

Semantyka przenoszenia
Przykłady

```
struct A
{
   int a, b;
};

A foo()
{
   return {1, 2};
}

A a;                        // l-value
A & ra = a;                 // l-value reference to l-value, OK
A & rb = foo();             // l-value reference to r-value, ERROR
A const& rc = foo();        // const l-value reference to r-value, OK (exception in rules)

A && rra = a;               // r-value reference to l-value, ERROR
A && rrb = foo();           // r-value reference to r-value, OK

A const ca{20, 40};
A const&& rrc = ca;         // const r-value reference to const l-value, ERROR
```

Semantyka przenoszenia

Konstruktor przenoszący oraz przenoszący operator przypisania

Zarówno konstruktor przenoszący jak i przenoszący operator przypisania jest generowany automatycznie przez kompilator, podobnie jak konstruktor kopiujący i kopiujący operator przypisania.

Domyślny konstruktor przenoszący przenosi każdą składową klasy.

Domyślny przenoszący operator przypisania oddelegowuje przeniesienie każdej składowej klasy do tego operatora zdefiniowanego dla tej składowej.

Semantyka przenoszenia

Przykład przenoszącego konstruktora i przenoszącego operatora przypisania

```
struct A
{
   A(A && src) :
       m_value(src.m_value)      // only example, can be much better
   {
      src.m_value.reset();
   }

   A & operator=(A && src)
   {
       m_value = src.m_value;    // only example, can be much better
       src.m_value.reset();
       return *this;
   }

   std::shared_ptr<int> m_value;
};
```

Semantyka przenoszenia
Nowe funkcje pomocnicze

std::move() - funkcja szablonowa przyjmująca r-value reference. Wykorzystuje zwijanie referencji (ang. *reference collapsing*) i wykonuje rzutowanie na referencję r-value. W przypadku przekazania l-value do funkcji algorytm dedukcji typu tworzy instancję szablonu przyjmującą referencję l-value, która jest następnie rzutowana na referencję r-value i dalej zwracana.

```
template <typename T>
typename std::remove_reference<T>::type&& move(T&& obj) noexcept
{
    using ReturnType = std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(obj);
}

A a;
A b = std::move(a);       // generates following template instance: A && move(A & obj) noexcept;
```

Semantyka przenoszenia
Przykład użycia std::move

```
struct A
{
   A(A && src) :
       m_value(std::move(src.m_value))
   {
   }

   A & operator=(A && src)
   {
       m_value = std::move(src.m_value);
       return *this;
   }

   std::shared_ptr<int> m_value;
};
```

Semantyka przenoszenia
Nowe funkcje pomocnicze

std::forward() przekazuje referencję do zmiennej. Funkcja szablonowa, podobnie jak std::move(), wykorzystuje zwijanie referencji, ale w przypadku gdy nastąpiło zwinięcie do referencji l-value zwraca referencję l-value, w przeciwnym wypadku referencję r-value.

Innymi słowy funkcja ta wykonuje tak zwany *perfect forwarding* czyli przekazuje dany parametr zachowując jego naturę r-value/l-value.

Semantyka przenoszenia
Przykład użycia std::forward

```
template<class Type>
class Bar
{
public:
   Bar(std::initializer_list<Type> values) :
        m_values(std::forward<std::initializer_list<Type>(values))       // much better
   {}

private:
   std::vector<Type> m_values;
};

Bar<int> b = { 1, 2, 5, 51 };
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- **Funkcje lambda**
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Funkcje lambda
Podstawowe funkcje lambda

Wyrażenie lambda jest definiowane najczęściej bezpośrednio”w miejscu” jego użycia (in-place). Zwykle jest użyte jako parametr innej funkcji, oczekującej wskaźnika do funkcji lub funktora - w ogólności obiektu wywoływalnego (callable object).

Każde wyrażenie lambda powoduje utworzenie przez kompilator unikalnej klasy domknięcia (closure class), która implementuje operator wywołania funkcji posiadający implementację użytą w wyrażeniu.

Domknięciem (closure) nazywana jest instancja klasy domknięcia. W zależności od sposobu przechwycenia zmiennych lokalnych obiekt ten przechowuje kopie lub referencje do przechwyconych zmiennych.

```
[](){}; // empty lambda

[] { std::cout << "hello world" << std::endl; } // unnamed lambda

auto l = [] (int x, int y) { return x + y; };

auto result = l(2, 3); // result = 5
```

Funkcje lambda
Podstawowe funkcje lambda

Jeśli implementacja lambdy nie zawiera instrukcji return typem zwracanym lambdy jest void.

Jeśli implementacja lambdy zawiera tylko instrukcję return typem zwracanym lambdy jest typ użytego wyrażenia

W każdym innym przypadku należy zadeklarować typ zwracany.

Wygodnie jest użyć lambd do tworzenia predykatów lub funktorów wymaganych przez algorytmy standardowe (na przykład w funkcji std::sort).

```
[](bool condition) -> int
{
    if (condition)
        return 1;
    else
        return 2;
}
```

```
std::array<double, 6> values = { 5.0, 4.0, -1.4, 7.9, -8.22, 0.4 };

std::sort(values.begin(), values.end(), [](double a, double b)
{
    return std::abs(a) < std::abs(b); // sortowanie wg wartości bezwzględnych
});
```

Funkcje lambda
Zakres zmiennych

Wewnątrz nawiasów kwadratowych [] możemy zawrzeć elementy, które lambda ma przechwycić z zakresu w którym jest tworzona oraz określić sposób w jaki zostaną one przechwycone.

•[ ] puste nawiasy oznaczają, że wewnątrz lambdy nie można użyć jakiejkolwiek nazwy z otaczającego kontekstu

•[&] niejawne przechwycenie przez referencję. Lambda ma dostęp do odczytu i zapisu zmiennych z zakresu w którym została utworzona. Obiekt domknięcia przechowuje referencje do zewnętrznych zmiennych.

•[=] niejawne przechwycenie przez wartość. Mogą być użyte wszystkie nazwy z zewnętrznego kontekstu. Nazwy te odnoszą się do kopii lokalnych zmiennych zewnętrznych. Ich wartość jest taka, jaka była w momencie tworzenia lambdy.

•[capture-list] jawne przechwycenie zmiennych wymienionych na liście. Domyślnie wymienione zmienne są przechwytywane przez wartość. Jeśli nazwy zmiennej jest poprzedzona przez & oznacza to przechwycenie przez referencję.

Funkcje lambda
Zakres zmiennych

```
#include <memory>
int a {5};
auto add5 = [=](int x) { return x + a; };

int counter {};
auto inc = [&counter] { counter++; }



int even_count = 0;
for_each(v.begin(), v.end(), [&even_count] (int n)
{
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});

cout << "There are " << even_count
     << " even numbers in the vector." << endl;
```

Funkcje lambda
Generyczne lambdy (C++14)

W C++11 parametry wyrażeń lambda musiały być zadeklarowane z użyciem konkretnego typu.

C++14 daje możliwość zadeklarowania typu parametru jako auto (*generic lambda*).

Powoduje to dedukcję typu parametru lambdy w ten sam sposób w jaki dedukowane są typy argumentów szablonu. W rezultacie kompilator generuje kod równoważny poniższej klasie domknięcia.

```
auto lambda = [](auto x, auto y) { return x + y; }


struct UnnamedClosureClass
{
    template <typename T1, typename T2>
    auto operator()(T1 x, T2 y) const
    {
        return x + y;
    }
};

auto lambda = UnnamedClosureClass();
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- **Wyrażenia *constexpr***
- Scoped enums
- Delegowanie konstruktorów
- Variadic templates

Constexpr

C++11 wprowadza dwa znaczenia dla “stałej”:

- constexpr - stała ewaluowana na etapie kompilacji
- const - stała, której wartość nie może ulec zmianie

Wyrażenie stałe (*constant expression*) jest wyrażeniem ewaluowanym przez kompilator na etapie kompilacji. Nie może zawierać wartości, które nie są znane na etapie kompilacji i nie może mieć efektów ubocznych.

Jeśli wyrażenie inicjalizujące dla constexpr nie będzie mogło być wyliczone na etapie kompilacji kompilator zgłosi błąd:

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1; // error: initializer is not a contant expression
constexpr int x4 = x2; // OK
```

Constexpr
constexpr w C++14

W C++14 zniesione zostały ograniczenia, które wymuszał C++11. Każda funkcja może zostać oznaczona jako constexpr, poza poniższymi wyjątkami:

- używa zmiennych statycznych lub thread_local,
- deklaruje zmienne bez inicjalizacji,
- jest wirtualna,
- wywołuje funkcje nie-constepr,
- używa nie-litealnych typów (wartości nieznane w czasie kompilacji),
- używa bloków kodu ASM,
- posiada bloki try-catch lub rzuca wyjątki

Constexpr
Examples

```
constexpr int foo(int bar)
{
    if(bar < 20)
    {
        return 4;
    }

    int k = 5;
    for(int i = 0; i < 54; ++i)
    {
        bar++;
    }

    if(bar > 51)
    {
        return bar + k;
    }

    return 1;
}
```

Constexpr
Examples

```
struct Point
{
constexpr Point(int x_, int y_)
   : x(foo(x_)), y(y_)
{}

int x, y;
};

constexpr Point a = { 1, 2 };
```

Constexpr
Zmienne constexpr

W C++11 constexpr przed zmienną definiuje ją jako stałą, która musi zostać zainicjowana wyrażeniem stałym.

Ważne: stała const w odróżnieniu od stałej constexpr nie musi być zainicjowana wyrażeniem stałym.

```
constexpr int x = 7;
constexpr auto prefix = "Data";
constexpr int n_x = factorial(x);
constexpr double pi = 3.1415;
constexpr double pi_2 = pi / 2;
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- **Scoped enums**
- Delegowanie konstruktorów
- Variadic templates

Scoped enums
enum class, enum struct

C++11 rozszerza typ enum o „zakresowy enum”. Ten typ ogranicza zakres zdefiniowanych stałych tylko do tych, które znajdują się w wyrażeniu enum i nie pozwala na niejawne konwersje do typu int.

```
enum Colors
{
    RED = 10,
    BLUE,
    GREEN
};

Colors a = RED;
int c = BLUE;

enum class Languages
{
    ENGLISH,
    GERMAN,
    POLISH
};

Languages d = Languages::ENGLISH;
//int e = Languages::ENGLISH;          // Not possible
int e = static_cast<int>(Languages::ENGLISH);
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- **Delegowanie konstruktorów**
- Variadic templates

Delegowanie konstruktorów

Od C++11 na liście inicjalizacyjnej konstruktora można wywołać inny konstruktor. Pozwala to uniknąć duplikacji kodu.

```
class Foo {
public:
    Foo() {
        // code to do A
    }
    Foo(int nValue): Foo() { // use Foo() default constructor to do A
        // code to B
    }
};
```

Agenda

- Wprowadzenie do standardu C++14
- Alias ‚using'
- Uniwersalny pusty wskaźnik *nullptr*
- Słowo kluczowe *auto*
- Lista inicjalizacyjna
- Słowa kluczowe *default* i *delete*
- Słowa kluczowe *override* i *final*
- Inteligentne wskaźniki
- Semantyka przenoszenia
- Funkcje lambda
- Wyrażenia *constexpr*
- Scoped enums
- Delegowanie konstruktorów
- **Variadic templates**

Variadic templates
Składnia

Szablony o zmiennej liczbie argumentów (ang. variadic templates) wykorzystują nową składnię grupowego parametru szablonu (ang. parameter pack), który reprezentuje wiele lub zero parametrów szablonu.

```
template<class... Types>
class variadic_class
{
/*...*/
};

template<class... Types>
void variadic_foo(Types&&... args)
{
/*...*/
}

variadic_class<float, int, std::string> v;

variadic_foo(1, "", 2u);
```

Variadic templates
Rozpakowywanie parametrów funkcji

Rozpakowywanie grupy parametrów wykorzystuje operator ... (elipsis).

W przypadku argumentów funkcji powoduje rozpakowanie podanych argumentów w kolejności ich podania.

Możliwe jest zawołanie funkcji na podanych parametrach. W tym przypadku podana funkcja (lub szablon funkcji) będzie wywołany na każdym argumencie z osobna.

Możliwe jest również wykorzystanie rekursji do rozpakowanie pojedynczych argumentów. Wymaga to zdefiniowania szablonu wariadycznego Head/Tail oraz funkcji nieszablonowej.

Variadic templates
Przykład

```
template<class... Types>
void variadic_foo(Types&&... args)
{
   callable(args...);
}


template<class... Types>
void variadic_perfect_forwarding(Types&&... args)
{
   callable(std::forward<Types>(args)...);
}

void variadic_foo() {}

template<class Head, class... Tail>
void variadic_foo(Head const& head, Tail const&... tail)
{
   /*action on head*/
   variadic_foo(tail...);
}
```

Variadic templates
Rozpakowywanie parametrów szablonów klas

Rozpakowywanie parametrów w szablonach klas wygląda analogicznie.

Możliwe jest rozpakowanie wszystkich typów (np. w przypadku klasy bazowej będącej szablonem o zmiennej liczbie parametrów) lub wykorzystanie specjalizacji częściowej i szczegółowej.

Variadic templates
Przykład

```
template<class... Types>
struct Base
{
};

template<class... Types>
struct Derived : Base<Types...>
{
};


template<int... Number>
struct Sum;

template<int Head, int... Tail>
struct Sum<Head, Tail...>
{
   const static int RESULT = Head + Sum<Tail...>::RESULT;
};

template<>
struct Sum<>
{
   const static int RESULT = 0;
}

Sum<1, 2, 3, 4, 5>::RESULT; // = 15
```

Variadic templates
operator sizeof...

Operator sizeof... pozwala odczytać liczbę parametrów w grupie na etapie kompilacji

```
template<class... Types>
struct NumOfArguments
{
   const static unsigned NUMBER_OF_PARAMETERS = sizeof...(Types);
};
```