



Zaawansowane programowanie w C++

Witold Wysota



Witold Wysota

Wykształcenie

Absolwent Politechniki
Warszawskiej

Programowanie

Specjalizacja: C++, Qt, architektura
Wykładowca, trener i konsultant

Działalność

Współtwórca i administrator
serwisu QtCentre.org
Qt in Education Advisory Board



Witold Wysota



Bibliografia

- *Undo/Redo with ItemViews*
- *Keeping the GUI Responsive*
- *Game Programming using Qt*

Politechnika Warszawska

Biblioteka Narodowa
Zenimax Online Studios

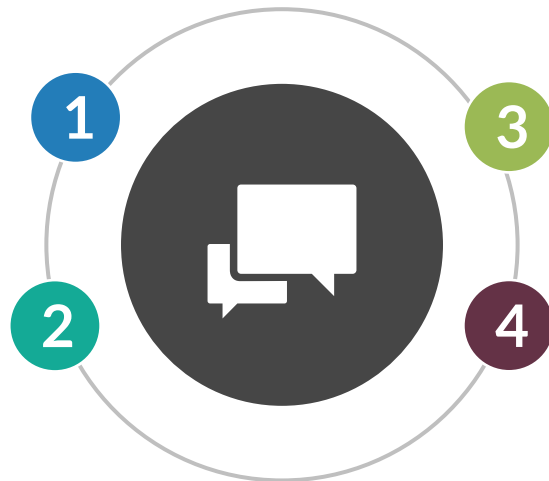
Plan szkolenia

Podstawy

Przegląd mechanizmów wprowadzonych w nowych standardach C++

Algorytmy i iteratory

Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



Szablony, metaprogramowanie

Programowanie na szablonach,
programowanie w czasie kompilacji

Pytania i podsumowanie

Wyjaśnienie nurtujących kwestii, odpowiedzi
na pytania

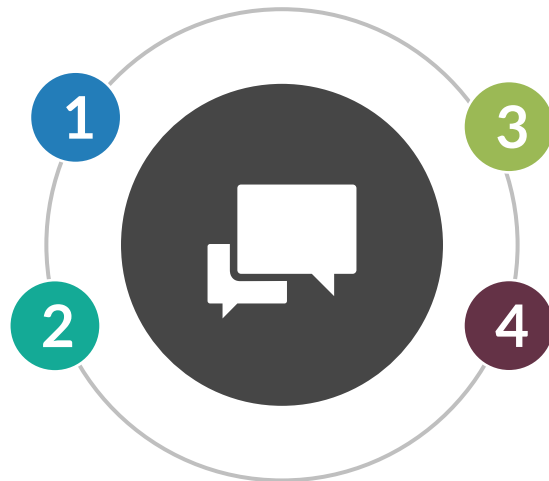
Plan szkolenia

Podstawy

Przegląd mechanizmów wprowadzonych w nowych standardach C++

Algorytmy i iteratory

Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



Szablony, metaprogramowanie

Programowanie na szablonach,
programowanie w czasie kompilacji

Pytania i podsumowanie

Wyjaśnienie nurtujących kwestii, odpowiedzi
na pytania

Zagadka

```
void test(const char *);  
void test(int);
```

```
int main() {  
    test(NULL);  
}
```

nullptr



- NULL
 - `#define NULL 0`
 - Możliwe inne implementacje (0L, nullptr, ...)
- nullptr
 - Wartość typu `nullptr_t`
 - Niejawnie konwertowalna do nullowego wskaźnika dowolnego typu
 - Jawnie oznacza nieprawidłowy wskaźnik a nie wartość '0' (typu `int` lub `long`)
 - Może zapobiec błędnemu wybraniu przeciążenia funkcji (`int` zamiast `typ*`)




Zagadka

```
typedef A B;
```

- Jaki typ zdefiniowaliśmy? A czy B?

using



- alias na istniejący typ
 - odpowiednik typedef z wcześniejszych standardów
 - `using stringvector = vector<string>;` 
 - pozwala na aliasowanie szablonów
 - `template <typename T1, typename T2> class K {};`
`template <typename T1> using K_int = K<T1, int>;` 
 - `template <typename T> class C {};`
`template <typename T> using mutable_C = C<std::remove_const<T>>;` 
- Składnia dużo przyjemniejsza niż typedef
- Aliasy szablonów nie podlegają mechanizmowi dedukcji
 - kompilator nigdy nie wydedukuje aliasu

Dziedziczenie metod

```
class Base {  
public:  
    void f(double, double) {}  
    void f(double) {}  
    void f(std::string) {}  
};  
  
class Derived : public Base {  
public:  
    void f(int) {}  
};  
  
Derived d;  
d.f(2.0); // która metoda się wywoła?
```

Dziedziczenie metod

- C++ dziedziczy całe rodziny metod (wszystkie kandydatury na przeciążenia)
- Jeżeli klasa pochodna zawiera metodę o tej samej nazwie, tworzy ona nową rodzinę metod
 - W związku z tym metody z klasy bazowej o tej nazwie nie są widoczne
- Istnieje możliwość “zaimportowania” rodziny metod z klasy bazowej

```
class Derived : public Base {  
public:  
    using Base::f;  
    void f(int) { }  
};
```

- Od C++11 da się w ten sposób (poprzez `using`) zaimportować konstruktory



Delegowanie konstruktorów

```
class Base {  
public:  
    Base(std::string) {}  
};
```

```
class Derived : public Base {  
public:  
    Derived(std::string s) : Base(s) {}  
};
```

- C++98 dopuszcza wywoływanie konstruktorów klasy bazowej w ramach listy inicjalizacyjnej konstruktora

Delegowanie konstruktorów



```
class Base {  
public:  
    Base(std::string) {}  
};
```

```
class Derived : public Base {  
public:  
    Derived(std::string s) : Base(s) {}  
    Derived(double) : Derived("s") {}  
};
```

- C++11 dopuszcza wywoływanie innych konstruktorów tej samej klasy
- Uwalnia od konieczności posiadania dedykowanej metody inicjalizacyjnej

Inicjalizacja



Zagadka - co to jest?

Klasa k1(7, 8);

Zagadka - co to jest?

```
Klasa k1(7, 8);
```

```
Klasa k2(7);
```

Zagadka - co to jest?

Klasa k1(7, 8);

Klasa k2(7);

Klasa k3();

Zagadka - co to jest?

Klasa k1(7, 8);

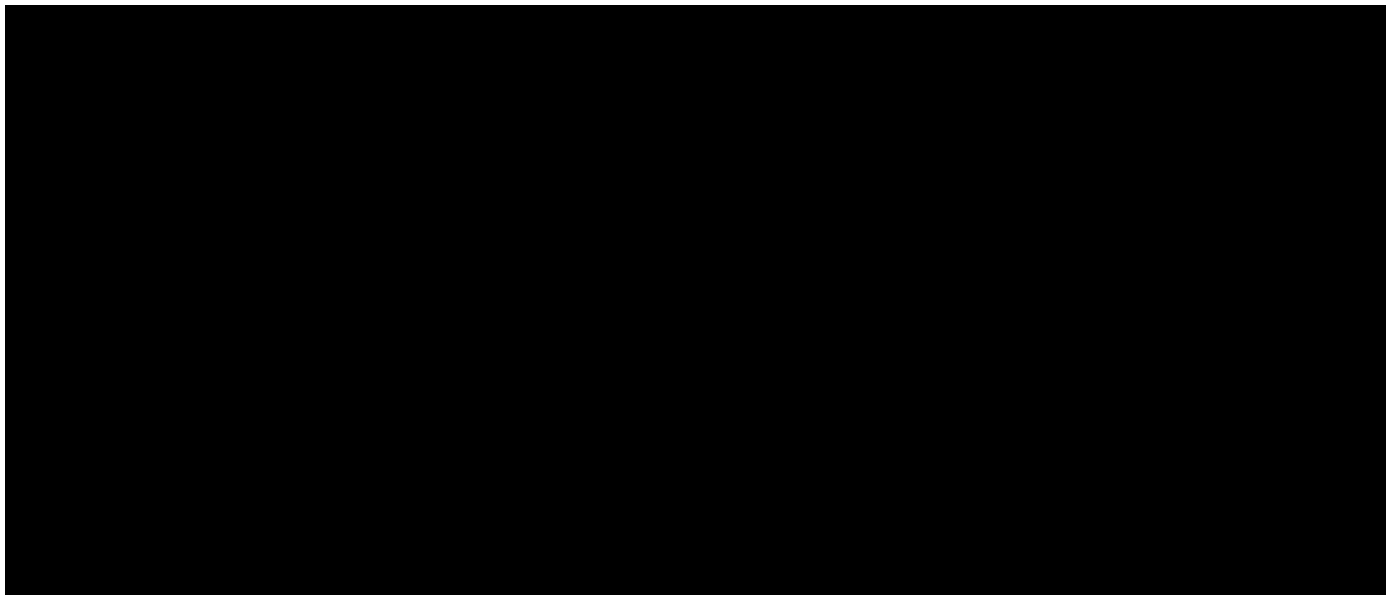
Klasa k2(7);

Klasa k3();

W C++ wszystko, co wygląda jak deklaracja, jest deklaracją.



Sposoby inicjalizacji w C++



<https://blog.tartanllama.xyz/initialization-is-bonkers/>



Podstawowe rodzaje inicjalizacji

```
// Default init:
// =====
// - if T is a class, the default ctor is called
// - if T is an array, each element is default-initiated
// - otherwise no init is done
int a                => [ 0 ] // uninitialized
Array                => [ 0, 0 ] // uninitialized
std::string s        => [  ]
```

Podstawowe rodzaje inicjalizacji

```
// Value init:
// =====
// - if T is a class then default-init, if no def-ctor then first zero-init
// - if T is an array, each element is value-init
// - otherwise zero-init
int b = 0, c{}           => [ 0, 0 ]
std::string{}            => [  ]
std::string{"abc"}       => [ abc ]
std::string("abc")       => [ abc ]
Array{}                  => [ 0, 0 ]
```

Podstawowe rodzaje inicjalizacji

```
// Zero init:  
// =====  
// - if T is a scalar, set to 0  
// - if T is an array, each element is zero-initiated  
// - if T is a class, all base-classes and members are zero-initiated
```

Inicjalizacja przez kopiowanie i wprost

```
// Copy init:
```

```
// -----
```

```
int a = 1                => [ 1 ]
```

```
int b = 2.3              => [ 2 ]
```

```
std::string s = "abc"    => [ abc ]
```

```
// Direct init:
```

```
// -----
```

```
int a(1), int b(2.3), int c{3} => [ 1, 2, 3 ]
```

```
int d{2.3}               // error: narrowing conversion of '2.3' from 'double' to 'int'
```

Inicjalizacja agregatów

```
// Aggregate initialization:
// -----
// struct S {
//     int i;
//     int j{-1};
// };
S s5 = {}           => [ 0, -1 ]
S s6 = {7, 8}       => [ 7, 8 ]
S s7 = {1}          => [ 1, -1 ]
S s8 = { {}, {} }   => [ 0, 0 ]
```

Określona inicjalizacja

```
// Designated initializers:
```

```
// -----
```

```
S s9 = {.i = 9}           => [ 9, -1 ]
```

```
S s10 = {.j = 8}          => [ 0, 8 ] // uninitialized
```

```
S s11 = {.i = 7, .j = 8}  => [ 7, 8 ]
```

```
S s12 = {.j = 9, .i = 8}  // error: designator order for field 'S::i' does not  
                           match declaration order
```


Inicjalizacja przez listę

Array	=> [848, 0] // uninitialized
Array{}	=> [0, 0]
Array = {}	=> [0, 0]
Array = {1}	=> [1, 0]
Array = {1, 2}	=> [1, 2]
Array{3, 4}	=> [3, 4]

Vector(5)	=> [0, 0, 0, 0, 0]
Vector(5, 2)	=> [2, 2, 2, 2, 2]
Vector{5, 2}	=> [5, 2]
vector<float> { 1, 2.1, 3.2f }	=> [1, 2.1, 3.2]
vector<int> { 1, 2.1, 3.2f }	// error: narrowing conversion...
vector<int> v = { 1, 2.1 }	// error: narrowing conversion...

Zmienne statyczne i globalne


```
// Static initialization (zero-init):  
// -----  
static struct { int i; }      => [ 0 ]  
static struct { int i = 4; } => [ 4 ]
```

- w ramach jednego pliku (jednostki translacyjnej) zmienne są inicjalizowane “od góry do dołu”
- kolejność inicjalizacji jednostek translacyjnych jest nieokreślona
- static initialization order fiasco
 - `static Foo f = g; // a.cpp`
`static Foo g = Foo{...}; // b.cpp`
 - jeśli a.cpp zainicjuje się przed b.cpp, to jest problem...
 - można obejść przez użycie funkcji ze zmienną statyczną
 - `Foo gFunc() { static Foo g{...}; return g; } // b.cpp`
`static Foo f = gFunc(); // a.cpp`

Lista inicjalizacyjna



std::initializer_list

- lekki obiekt, dający dostęp do listy elementów określonego typu
 - kopiowanie listy nie kopiuje jej zawartości
 - czas życia kopii przedłuża czas życia danych 
 - tworzone przy użyciu nawiasów klamrowych
 - { 1, 2, 3, 4, 5 }
 - { "abc", "def", "ghi", "jkl" }
- używane w kontekście konstruktorów i funkcji
 - dana funkcja przyjmuje `std::initializer_list<T>`
- używane w kontekście definiowania zakresu dla for



Lista inicjalizacyjna

Zainicjuj przy pomocy `initializer_list` strukturę danych przechowującą daty urodzin grupy ludzi.



Metody generowane przez kompilator

- Konstruktor domyślny
 - Jeżeli nie zadeklarowano żadnego konstruktora
- Trywialny destruktor
- Konstruktor kopiujący
 - Gdy nie zadeklarowano operatora lub konstruktora przenoszącego
- Operator przypisania
 - jak wyżej
- Konstruktor przenoszący
 - Gdy nie zadeklarowano destruktor, operatora ani konstruktora kopiującego, operatora przeniesienia
- Operator przeniesienia
 - Gdy nie zadeklarowano destruktor, operatora ani konstruktora przypisania, konstruktora przenoszącego

Zasada 0

- Jeśli możesz uniknąć definiowania metod specjalnych, to ich nie definiuj
 - jeśli typy składowych są poprawnie zdefiniowane, to kompilator zrobi, co trzeba
 - dotyczy to również (jeśli nie przede wszystkim) **destruktor**a
 - implementując (lub deklarując) niepotrzebnie metody specjalne, możesz negatywnie wpłynąć na wydajność

Zasada 5

- jeśli implementujesz jedną z metod specjalnych, zrób to też dla pozostałych
 - metody specjalne mają ze sobą dużo wspólnego
 - jeśli domyślne implementacje (np. konstruktora) nie wystarczają, prawdopodobnie pozostałe metody też trzeba zmienić

Klasa niekopiowalna

```
class Niekopiowalna {  
public:  
    Niekopiowalna();  
private:  
    Niekopiowalna(const Niekopiowalna &);  
    Niekopiowalna& operator=(const Niekopiowalna &);  
};
```

- Konstruktor kopiujący jest prywatny
- Nikt nie może utworzyć obiektu na podstawie takiego samego
- Dla porządku prywatny powinien być też operator przypisania

Klasa niekopiowalna



```
class Niekopiowalna {  
public:  
    Niekopiowalna();  
    Niekopiowalna(const Niekopiowalna &) = delete;  
    Niekopiowalna& operator=(const Niekopiowalna &) = delete;  
};
```

- Metody, których nie chcemy są explicitie oznaczone jako nieistniejące

Default, delete



- Dowolną metodę lub funkcję możemy oznaczyć jako skasowaną (= delete)
- Metody, które może wygenerować kompilator możemy oznaczyć jako domyślne (= default)
- Kod staje się bardziej czytelny
- Nie trzeba implementować ciała trywialnego konstruktora lub destruktora
- Możemy jawnie zabronić pewnych wywołań (np. specjalizacji szablonu)
 - `template<class U, class V> V convert(U) { ... }`
`template<> X convert(Y) = delete;`

Kontrola na poziomie kompilacji

- delete
- explicit
 - konstruktor konwertujący
 - konstruktor kopiujący
 - operatory konwersji
- specjalizacje szablonów
 - delete
 - wystawienie innego (lub żadnego) API
 - koncepcje, SFINAE
 - asercja
- kontrola na poziomie kompilacji jest lepsza niż niezadeklarowanie metody
 - bo zaraz ktoś przyjdzie i ją dopisze...
 - i nie doda nas do review...

static_assert



- `assert()` dokonuje sprawdzenia w trakcie działania programu
- `static_assert()` dokonuje ewaluacji podczas kompilacji
 - `static_assert(sizeof(N) > 2, "N musi być większe niż 2")`
 - `static_assert(std::is_same(typename U::type, int), "U::type musi być typu int");`
- Warunek musi być weryfikowalny podczas kompilacji
- Od C++14 można pominąć komunikat o błędzie


std::function

C++11

```
std::function<bool(int, int)> funkcja; // funkcja zwracająca bool i pobierająca int, int
    if (!funkcja)
        funkcja = [](int a, int b) { return a+b < 4; };
    bool r = funkcja(3, 4);
struct K {
    K(){}
    void memFun(int arg) { std::cout << __FUNCTION__ << ": " << arg << std::endl; }
    double n;
};
std::function<void(K&, int)> funkcja = &K::memFun; // funkcja pobierająca K& oraz int
K k;
funkcja(k, 42);
std::function<double(K&)> funkcja2 = &K::n; // funkcja pobierająca K& oraz int
double n = funkcja2(k);

funkcja = nullptr;      // wyzerowanie funkcji
```

std::function

- owijka na funkcję
- Obsługiwane typy
 - Funkcja
 - Wyrażenie lambda i inne obiekty funkcyjne
 - Wskaźniki na metody klasy
 - Wskaźniki na pola klasy
 - Ogólnie na wszystko co da się zawołać przez std::invoke() 
- Pozwala przechowywać, kopiować, wołać
- Próba wywołania pustego obiektu kończy się wyjątkiem std::bad_function_call

Kategorie wartości



kategorie wartości

W dużym uproszczeniu:

- **lvalue** - coś, co posiada adres
 - funkcja
 - obiekt
- **rvalue** - coś, co nie posiada adresu
- **xvalue** - obiekt kończący swój żywot
 - jego zasoby mogą zostać wykorzystane
- **prvalue** - rvalue, który nie jest xvalue



rvalue

- *right value* - historycznie wszystko co nie może się pojawić po lewej stronie operatora przypisania
 - obiekty tymczasowe
 - zwracane przez funkcje
 - związane z automatyczną konwersją typów
 - większość literałów (coś, co nie jest obiektem)
 - xvalue
- do rvalue można przypisać wyłącznie wyrażenia rvalue

lvalue

- *left value* - historycznie wszystko co może się pojawić po lewej stronie operatora przypisania
 - Jeżeli da się pobrać adres wyrażenia, to wyrażenie to jest lvalue
 - zmienne, funkcje, pola w klasie
 - Jeżeli typ wyrażenia jest referencją na lvalue, to samo wyrażenie też jest lvalue
- do lvalue można przypisać zarówno wyrażenia lvalue jak i rvalue

xvalue, glvalue, prvalue



- *expring value*
 - obiekt najczęściej pod koniec swojego życia, którego zasoby mogą zostać zabrane (wykorzystane)
 - np. wynik wywołania funkcji, której zwracany typ jest referencją na rvalue
 - `std::move(x)`
 - wskaźnik na składową obiektu, jeżeli obiekt jest rvalue
 - operator warunkowy
 - `a?b:c`
- *generalized lvalue*
- *pure rvalue*
 - np. wynik wywołania funkcji, której zwracany typ nie jest referencją

Referencje Rvalue



- Pozwalają przedłużyć czas życia obiektów tymczasowych
 - `std::string str = "abc";`
`std::string && twice = str + str;`
- Do funkcji można dodać przeciążenia gdzie jedno wiąże do lvalue ref a drugie do rvalue ref
- Wykorzystanie w specjalny sposób obiektów, których nikt już nie potrzebuje
- Przed C++11: Kompilator stosował tę strategię niejawnie (copy elision)
- Od C++11: Możliwość użycia `explicit`

Zamiana na r-value reference



- Wybranie przez kompilator wariantu funkcji z referencją rvalue wymaga wartości rvalue
 - tylko rvalue można przypisać do rvalue

```
void fun(std::vector<int> && v) {}  
std::vector<int> vec = { 1, 2, 3, 4, 5 };  
fun(vec);
```

error: cannot bind rvalue reference of type '`std::vector<int>&&`' to lvalue of type '`std::vector<int>`'

- “Zwykłe” zmienne są przeważnie typu lvalue
- Standard wprowadza funkcję pozwalającą zamienić lvalue na rvalue

```
std::vector<int> vec = { 1, 2, 3, 4 };           // lvalue  
std::vector<int> && rvec = std::move(vec);       // rvalue
```

- Uwaga! `std::move()` nic nie przenosi i nie robi żadnej innej magii, po prostu zamienia na `&&`

Funkcje operujące na &&



- Znaczenie parametrów &&: “nikt tego nie używa, zrób z tym, co chcesz”
- Możliwość zabrania zawartości obiektu
- Obiekt powinno się zostawić w stanie “używalnym”

```
void fun(std::vector<int> && v) {  
    std::vector<int> tmp;  
    std::swap(tmp, v);  
}
```

```
std::vector<int> vec = { 1, 2, 3, 4, 5 };  
fun(std::move(vec));  
std::cout << vec.size() << std::endl; // 0
```

Konstruktor przenoszący



- `K(const K& other)`
 - Utwórz mi obiekt identyczny z `other`, pozostaw `other` niezmienione (`const`)
- `K(K&& other)`
 - Utwórz mi obiekt wykorzystując `other`, którego nikt nie potrzebuje, możesz wykorzystać jego dane
- Dwie możliwości
 - Definiowany obok konstruktora kopiującego
 - wydajniejsze stworzenie obiektu w pewnych warunkach
 - Definiowany zamiast konstruktora kopiującego
 - gwarantuje zabranie danych z pierwotnego obiektu

Operator przeniesienia



- `K& operator=(const K& other)`
 - Zrób ze mnie kopię `other`, pozostaw `other` niezmienionym
- `K& operator=(K&& other)`
 - Zrób ze mnie `other`, wykorzystaj jego dane jeśli to możliwe, bo on zaraz umrze
- Dwie możliwości
 - Definiowany obok operatora przypisania
 - Zwiększenie wydajności przypisania jeśli ktoś powie, że można
 - Definiowany zamiast operatora przypisania
 - Przekazanie własności danych pomiędzy obiektami


```
struct Item {  
    string name;  
    double value;  
};  
  
struct Broadsword : Item {};  
struct Shield : Item {};  
  
double totalValue(const vector<Item *> &container);  
  
void foo() {  
    vector<Item *> inventory;  
    inventory.push_back(new Broadsword);  
    inventory.push_back(new Shield);  
    cout << totalValue(inventory) << endl;  
}
```

```
void foo() {  
    vector<Item *> inventory;  
    inventory.push_back(new Broadsword);  
    inventory.push_back(new Shield);  
    cout << totalValue(inventory) << endl;  
    // release memory  
    for (auto *p : inventory) {  
        delete p;  
    }  
}
```

```
void foo() {  
    vector<Item *> inventory;  
    inventory.push_back(new Broadsword);  
    inventory.push_back(new Shield);  
    cout << totalValue(inventory) << endl;  
    // release memory  
    for (auto *p : inventory) {  
        delete p;  
    }  
}
```

← może rzucić wyjątek!

```
void foo() {  
    vector<Item *> inventory;  
    inventory.push_back(new Broadsword);  
    inventory.push_back(new Shield);  
    cout << totalValue(inventory) << endl;  
    // release memory  
    for (auto *p : inventory) {  
        delete p;  
    }  
}
```

wyciek pamięci

może rzucić wyjątek!

```
void foo() {  
    vector<Item *> inventory;  
    inventory.push_back(new Broadsword);  
    inventory.push_back(new Shield);  
    cout << totalValue(inventory) << endl;  
    // release memory  
    for (auto *p : inventory) {  
        delete p;  
    }  
}
```

wyciek pamięci

może rzucić wyjątek!

The diagram consists of two red arrows pointing from the text 'wyciek pamięci' to the lines 'inventory.push_back(new Broadsword);' and 'inventory.push_back(new Shield);'. A second red arrow points from the text 'może rzucić wyjątek!' to the line 'cout << totalValue(inventory) << endl;'. This illustrates that manually deleting elements from a container can lead to memory leaks and that the totalValue function might throw an exception.

Właściciel odpowiada za usunięcie

```
User* getUser(const string &name);  
bool logout(User* user);
```

- Czy te funkcje przekazują własność obiektu User?
- Czy getUser tworzy nowy obiekt czy tylko zwraca już istniejący?
- Jaki jest stan obiektu po zwołaniu logout?

- czasami informacja zaszyta jest w dokumentacji lub w kodzie źródłowym

```
User* getUser(const string &name); // transfers ownership to caller
```

Inteligentny wskaźnik

- komitet standaryzacyjny C++ zaleca, żeby do śledzenia własności danych ze sterty używać inteligentnych wskaźników
- obiekt, który zachowuje się jak wskaźnik
 - operator* wyłuskuje dane
 - `auto data = *ptr;`
 - operator-> zwraca referencję na dane
 - `ptr->performAction();`
 - można przypisać do niego nullptr
 - `ptr = nullptr;`
 - można sprawdzić, czy zawiera nienullową wartość
 - `if (ptr) { ... }`

std::unique_ptr<T>

- zawsze jest dokładnie jeden właściciel danych (stąd nazwa)
- zniszczenie wskaźnika usuwa posiadane przezeń dane (RAII)
- zrobienie kopii wskaźnika niedozwolone

```
#include <memory>
```

```
void foo() {  
    vector<std::unique_ptr<Item> > inventory;  
    inventory.push_back(std::make_unique<Broadsword>());  
    inventory.push_back(std::make_unique<Shield>());  
    cout << totalValue(inventory) << endl;  
}
```


Tworzenie

- przypisanie danych do wskaźnika może nastąpić wyłącznie explicite

```
unique_ptr<X> ptr = new X;  
auto ptr = unique_ptr<X>(new X);  
unique_ptr<X> ptr(new X);
```

- od C++14 istnieje uniwersalna fabryka wskaźników (przeoczenie w C++11)
 - do funkcji przekazujemy argumenty konstruktora typu X

```
auto ptr = std::make_unique<X>(arg1, arg2, ...);
```

Tablice

- do wskaźnika może być przypisana cała tablica elementów

```
std::unique_ptr<X[]> tab = std::make_unique<X[]>(5);  
tab[1] = ...;
```

Przekazanie własności

własność danych może zostać przekazana z jednego wskaźnika do innego

```
std::unique_ptr<X> ptr = std::make_unique<X>();  
std::unique_ptr<X> other = ptr;  
std::unique_ptr<X> other = std::move(ptr);  
assert(ptr == nullptr);
```

- zawsze dokładnie jeden wskaźnik jest właścicielem danych
- nie ma momentu, żeby dane były bez właściciela
- po przeniesieniu w starym wskaźniku nie ma danych (nullptr)
- nie ma fizycznie możliwości wykonania “kopii” wskaźnika

unique_ptr<T> w praktyce

- `std::unique_ptr<User> getUser(const std::string &name);`
 - jawnie pokazuje, że wołający jest odpowiedzialny za zwolnienie obiektu
 - zapobiega błędom zignorowania wartości zwracanej z funkcji
 - automatycznie zwolni pamięć po wyjściu z zakresu

`auto user = getUser("maurycy"); // std::move nie jest potrzebne`

- `bool logout(std::unique_ptr<User> user);`
 - jawnie pokazuje, że funkcja przejmuje obiekt na własność
 - wołający traci dostęp do obiektu
 - funkcja gwarantuje, że zwolni obiekt użytkownika

Jak nie przekazać własności?

- unique_ptr wykorzystywany tylko do modelowania własności
- przy braku przekazania własności używamy...

- zwykłego wskaźnika

```
bool logout(User *user);  
std::unique_ptr<User> ptr = ...;  
logout(ptr.get());
```

- referencji

```
bool logout(User &user);  
std::unique_ptr<User> ptr = ...;  
if (ptr) { logout(*ptr); }
```



Koszyk sklepowy

1. Uzupełnij implementację koszyka sklepowego.
2. Unowocześnij kod, używając inteligentnych wskaźników.



Zwalnianie pamięci

- domyślnie `unique_ptr` niszczy obiekt poprzez operator `delete`
- własny sposób usuwania
 - operator `delete` się nie nadaje (np. dane pochodzą z C)
 - obiekt wymaga specjalnego potraktowania przy usuwaniu
 - chcemy zrobić coś innego zamiast zniszczenia obiektu (np. oddać do puli)
- wskaźniki z różnymi rodzajami usuwania są niekompatybilne
 - podajemy typ funkcji usuwającej jako drugi parametr szablony

Usuwanie - wskaźnik do funkcji



```
api* api_init();  
void api_free(api *);
```

```
using ApiPtr = std::unique_ptr<api, void(*) (api*)>;
```

```
ApiPtr createApi() {  
    return ApiPtr(api_init(), api_free);  
}
```


Ustawianie - funktor



```
api* api_init();  
void api_free(api *);
```

```
struct ApiDeleter {  
    void operator()(api *a) { api_free(a); }  
};
```

```
using ApiPtr = std::unique_ptr<api, ApiDeleter>;
```

```
ApiPtr createApi() { return ApiPtr(api_init()); }
```

std::shared_ptr<T>



- zawsze jest przynajmniej jeden współwłaściciel danych
- zrobienie kopii współdzieli własność
- zniszczenie ostatniego współwłaściciela zwalnia pamięć

```
#include <memory>
```

```
void foo() {  
    vector<shared_ptr<Product> > shoppingCart;  
    shoppingCart.push_back(make_shared<Bread>());  
    shoppingCart.push_back(make_shared<Butter>());  
    cout << totalPrice(shoppingCart) << endl;  
}
```

Tworzenie



- przypisanie danych do wskaźnika może nastąpić wyłącznie explicite

```
shared_ptr<X> ptr = new X;  
auto ptr = shared_ptr<X>(new X);  
shared_ptr<X> ptr(new X);
```

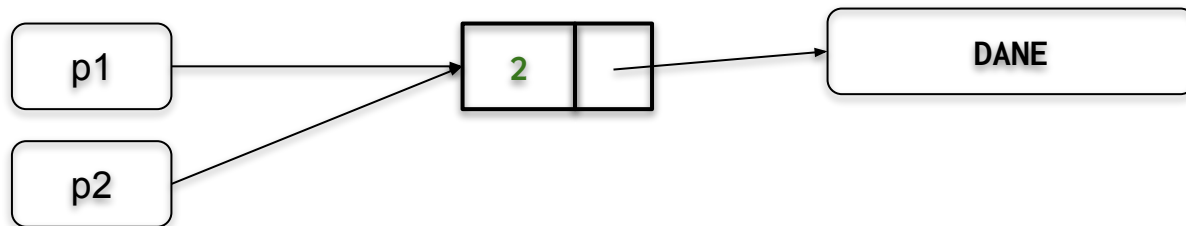
- od C++11 istnieje uniwersalna fabryka wskaźników

```
auto ptr = std::make_shared<X>(arg1, arg2, ...);
```

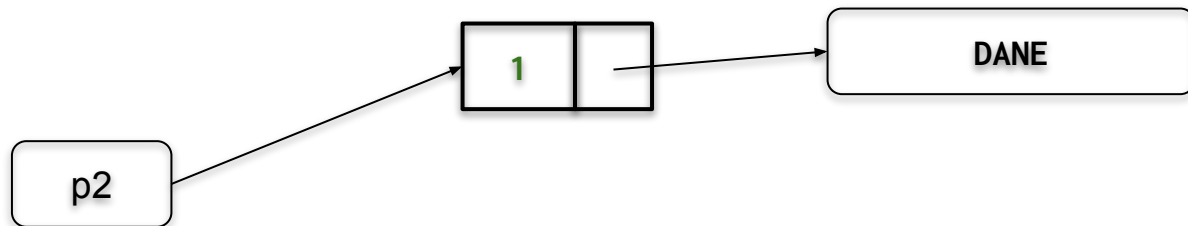
Blok kontrolny



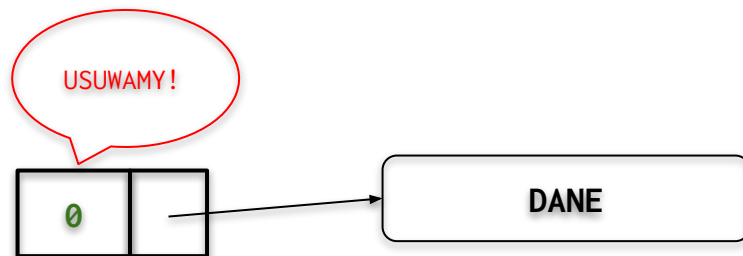
Blok kontrolny



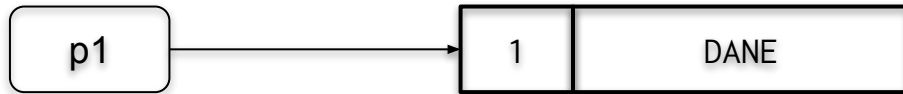
Blok kontrolny



Blok kontrolny



`std::make_shared<T>(...)`



Funkcje

- API podobne do tego z `unique_ptr` (`reset`, `release`)
- własne mechanizmy zwalniania pamięci
- `shared_ptr` jest kopiowalny
- `use_count()`
 - liczba referencji na dane
 - w środowisku wielowątkowym nie należy ufać tej wartości

Promocja z unique_ptr



```
unique_ptr<X> unique = make_unique<X>();  
shared_ptr<X> shared = move(unique);
```



shared_ptr



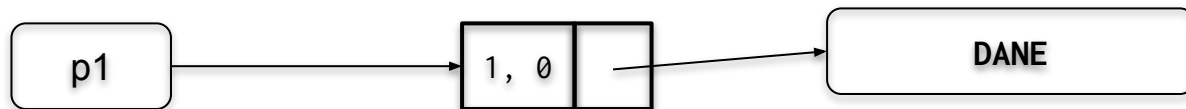
std::weak_ptr<T>



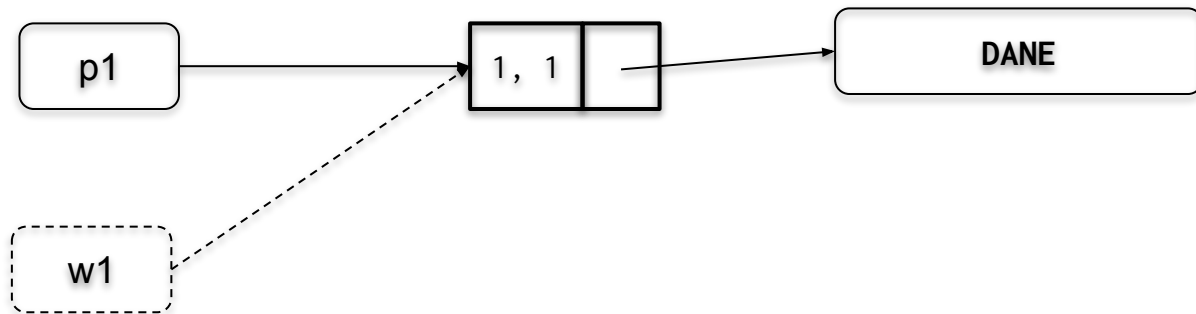
- tworzony z shared_ptr
- nie podbija liczby referencji na dane
- nie daje bezpośredniego dostępu do danych (*, ->)
- pozwala ponownie uzyskać shared_ptr

```
struct Person { std::vector<std::shared_ptr<class Object>> inventory; };  
struct Object {  
    std::weak_ptr<Person> owner;  
};
```

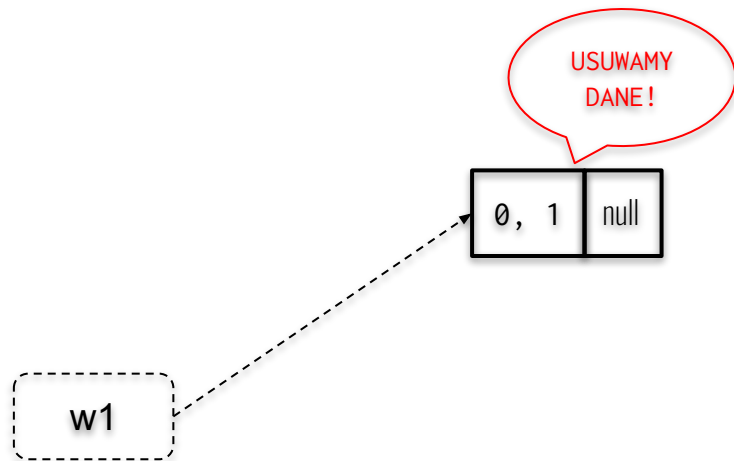
Blok kontrolny



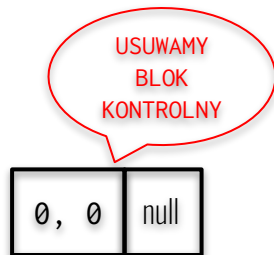
Blok kontrolny



Blok kontrolny



Blok kontrolny



Użycie weak_ptr<T>



```
std::shared_ptr<Person> matt = std::make_shared<Person>();  
std::weak_ptr<Person> w_matt = matt;
```

```
// ...
```

```
std::shared_ptr<Person> matt = w_matt.lock(); // podbija referencję - uda się lub nie  
if (matt) {  
    matt->...  
}
```

Dobry model obiektów w ogóle **nie dopuści** do (hipotetycznego) istnienia cyklu - weak_ptr nie jest tu rozwiązaniem tylko maskuje faktyczny problem

Praktyczne użycie weak_ptr<T>

```
std::shared_ptr<User> getUser(const std::string &name) {  
    static std::map<std::string, std::weak_ptr<User>> cache;  
    auto &ptr = cache[name];  
    auto shared = ptr.lock();  
    if (!shared) {  
        shared = std::make_shared<User>(name);  
        ptr = shared;  
    }  
    return shared;  
}
```

unique_ptr czy shared_ptr?

Skoro shared_ptr sam wszystkiego pilnuje to najlepiej używać go wszędzie i zapomnieć o zarządzaniu pamięcią.

Prawda...?



unique_ptr czy shared_ptr?

```
auto process = [&] {  
    for (size_t i = 0; i < iterations / thread_count; ++i) {  
        for (auto &ptr : container) {  
            ptr->doSomething();  
        }  
    }  
};
```

```
auto process = [&] {  
    for (size_t i = 0; i < iterations / thread_count; ++i) {  
        for (auto ptr : container) {  
            ptr->doSomething();  
        }  
    }  
};
```

1k obiektów, 1M iteracji, 1 wątek:

(unique_ptr)	Result: 12 ms
(shared_ptr reference)	Result: 11 ms
(shared_ptr copy)	Result: 10270 ms

1k obiektów, 1M iteracji, 10 wątków:

(unique_ptr)	Result: 3 ms
(shared_ptr reference)	Result: 4 ms
(shared_ptr copy)	Result: 3317 ms

owner<T>

```
template <class T, class = typename std::enable_if<std::is_pointer<T>::value>::type>  
using owner = T;
```

```
bool logout(owner<User*> user);
```

- jawne oznaczenie (konwencja), że funkcja przejmuje obiekt na własność
- kompilator dalej widzi to jako goły wskaźnik
- możliwość wychwycenia błędów narzędziami do statycznej analizy kodu

not_null<T*>

- wskaźnik, który nie dopuszcza do wartości nullptr
- podanie nullptr zasygnalizuje błąd lub spowoduje błąd kompilacji
- nie trzeba ręcznie sprawdzać wartości
- implementacja np. w bibliotece GSL (github.com/microsoft/GSL)

```
bool logout(not_null<User*> user) {  
    user->last_seen = std::chrono::system_clock::now();  
    return true;  
}
```


Dedukcja



Dedukcja typu zmiennej



Dedukcja typów

auto var = <wyrażenie>

- używane gdy typ wyniku wyrażenia jest
 - nieistotny
 - trudny do odgadnięcia
 - niewygodny do zapisania
 - słowu kluczowemu auto mogą towarzyszyć modyfikatory (const, &, *, itd.) pomagające w dedukcji
-
- `for (auto iter = container.begin(); iter != container.end(); ++iter) { ... }`
 - `auto lambda = ... { ... }`

Dedukcja typu zmiennej



Dedukcja typów

`decltype(<wyrażenie>) var;`

- dosłownie “typ taki jak wynik <wyrażenia>”
- wyrażenie nie jest wykonywane, sprawdzany jest tylko typ jego wyniku
 - zmienna
 - wywołanie operatora
 - wywołanie funkcji
- przydatne gdy chcemy mieć dedukcję typu, ale bez inicjalizacji

```
decltype(f(0)) var;  
// ...  
var = f(42);
```

Dedukcja typu wyniku funkcji



Dedukcja typów

```
auto funkcja(...) -> decltype(<wyrażenie>) { ... }
```

- auto nie dokonuje dedukcji typu
- typ wartości zwracany przez funkcję określany jest za pomocą decltype

```
auto findElem(const std::vector<T>& v, size_t which) -> decltype(v.begin()) {  
    if (which < v.size())  
        return v.begin()+which;  
    return v.end();  
}
```

Dedukcja typu wyniku funkcji



Dedukcja typów

auto funkcja(...) { ... }

- typ funkcji określany jest na podstawie operandu **return**
- ewentualne wywołania rekurencyjne nie mogą być przed pierwszym **return**
- wszystkie **return** muszą zwracać ten sam typ

```
auto funkcja(int a, int b) {  
    if (a < 0 && b < 0)  
        return std::min(a, b);  
    if (a > 0 && b < 0)  
        return a + b;  
    return std::max(a, b);  
}
```

Dedukcja typu klasy szablonowej



```
std::pair p = {1, 2.0};
```

- Dedukcja parametrów klasy szablonowej na podstawie wywołania konstruktora
- Niejasny zapis typu
 - `std::pair<int, double>` vs `std::pair`
- Możliwy zapis:
 - `std::pair p = {1, 2.0};`

Almost-Always-Auto

Jak używać auto

- Preferowanie dedukowanych typów nad jawnie określonymi
- Używanie auto, decltype, decltype(auto)
- Tworzenie kodu dopasowanego do interfejsu a nie implementacji

- Jeżeli typ ma podążać za zmianami w kodzie, polegamy na dedukcji

```
auto a = init;
```

- Jeżeli typ ma być niezmienny, polegamy na jawnym określeniu

```
auto b = Type{ init };
```

```
Type c{ init };
```



Uniwersalne referencje

Gdy zmienna lub parametr jest zadeklarowana jako typ **T&&** dla pewnego **dedukowanego** typu T, zmienna lub parametr jest określana mianem *uniwersalnej referencji*.

- Rzeczywisty typ uniwersalnej referencji zależy od wyrażenia, którym jest inicjowana
 - referencja na lvalue gdy inicjowana przez lvalue
 - referencja na rvalue gdy inicjowana przez rvalue
- Wynika to wprost z zasad sklejania referencji
 - **A& &** => **A&**
 - **A& &&** => **A&**
 - **A&& &** => **A&**
 - **A&& &&** => **A&&**



std::forward

Uniwersalne referencje

```
template<class S> S&& forward(typename remove_reference<S>::type& a) {  
    return static_cast<S&&>(a);  
}
```

std::forward<int>

Uniwersalne referencje

```
template<int> int&& forward(int& a) {  
    return static_cast<int&&>(a);  
}
```

std::forward<int&>

Uniwersalne referencje

```
template<int&> (int&)&& forward(int& a) {  
    return static_cast<(int&)&&>(a);  
}
```



std::forward<int&>

Uniwersalne referencje

```
template<int&> int& forward(int& a) {  
    return static_cast<int&>(a);  
}
```

std::forward<int&&>

Uniwersalne referencje

```
template<int&&> (int&&)&& forward(int& a) {  
    return static_cast<(int&&)&&>(a);  
}
```



std::forward<int&&>

Uniwersalne referencje

```
template<int&&> int&& forward(int& a) {  
    return static_cast<int&&>(a);  
}
```

Wyrażenia lambda



Wyrażenia lambda



Wyrażenie lambda definiuje *domknięcie* - anonimową funkcję, która przechowuje dane ze swojego środowiska, które są jej potrzebne do obliczeń

- Można ją przypisać do zmiennej (najczęściej auto)
- Można jej używać jak funkcji
- Realizacja przez obiekt funkcyjny (lambda to klasa a nie funkcja)

Lambda - składnia

[przechwycenia] (parametry) { ciało funkcji }

```
auto fun = [] (int a, int b) { return a > b; };
```

```
bool res = fun(2, 1);
```

```
std::vector<int> vec;
```

```
int cnt = std::count_if(vec.begin(), vec.end(), [](int v) { return v < 0; });
```

Lambda - przechwycenia

```
[przechwycenia] ( parametry ) { ciało funkcji }
```

Przechwycenia:

- = - wszystko przez wartość (kopia)
- & - wszystko przez referencję
- nazwa_zmiennej - pojedyncza zmienna przez wartość
- &nazwa_zmiennej - pojedyncza zmienna przez referencję
- definicje rozdzielone przecinkami
 - [&x, =] - x przez referencję, wszystko inne przez wartość
 - [this] - this przez wartość (od C++20 '=' nie obejmuje 'this')

Przechwycenia - przykład

```
unsigned val = 1;  
std::generate_n(output.begin(), 10, [&val]() { return val++; });
```

Lambda - realizacja

```
auto lambda = [x, y, &z](int a, int b) {  
    return x+y+z-a-b;  
};  
lambda(1, 2);
```

```
class FunObj {  
public:  
    FunObj(int _x, int _y, int &_z)  
        : x(_x), y(_y), z(_z) {}  
    int operator()(int a, int b) const {  
        return x+y+z-a-b;  
    }  
private:  
    int x, y, &z;  
};
```

```
auto funobj = FunObj(x, y, z);  
funobj(1, 2);
```



Lambda - Przechwytywanie wyrażeń 14

- C++11
 - [=,&v], [v,&], ... przechwytywanie przez wartość lub referencję istniejących zmiennych
- C++14
 - [&r = z, x = 7, ptr = std::move(o)]
 - deklaruje zmienne jako auto
 - przypisuje im wyniki wyrażeń

```
std::unique_ptr<Object> ptr = ...;
std::function<void(void)> f = [ o = std::move(ptr) ] () { consume(o.get()); };
// ...
f();
```

Generalizacja wyrażeń lambda



Dedukcja typów

```
[] (auto v1, auto v2) { ... }
```

- zasadniczo działa identycznie jak szablon:
 - `template<class U, class T1, class T2> U funkcja(T1 v1, T2 v2)`
- pozwala na stworzenie rodziny funkcji, które działają dla dowolnych danych zgodnych ze składnią ciała funkcji

```
auto l = [] (auto v1, auto v2) { return std::min(v1, v2); }
```

- `l` można zwołać na czymkolwiek, co posiada przeładowanie `std::min()`

Lambda w C++20

[przechwycenia] <specyfikacja typów> (parametry) { ciało funkcji }

```
auto min_cpp17 = [](auto left, auto right) { // left i right mogą być różnych typów
    return left < right ? left : right;
};
```

```
auto min_cpp20 = [] <typename T> (T left, T right) { // left i right są tego samego typu
    return left < right ? left : right;
};
```

auto w C++20

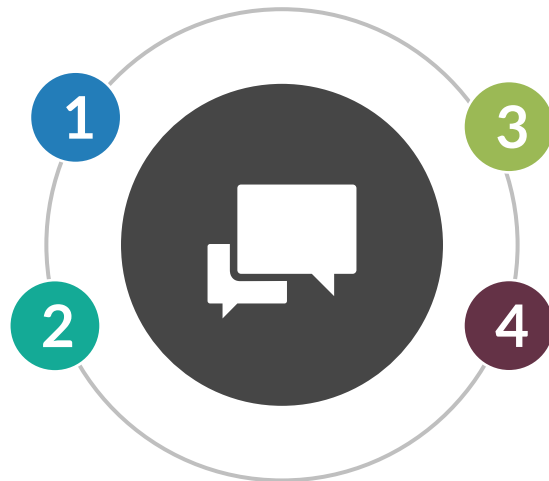
Plan szkolenia

Podstawy

Przegląd mechanizmów wprowadzonych w nowych standardach C++

Algorytmy i iteratory

Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



Szablony, metaprogramowanie

Programowanie na szablonych,
programowanie w czasie kompilacji

Pytania i podsumowanie

Wyjaśnienie nurtujących kwestii, odpowiedzi
na pytania

Co to jest za algorytm?

```
void algorithm1(int n, int t[])
{
    for(int i=0; i<n; i++) {
        int k=i;
        for(int j=i+1; j<n; j++)
            if(t[j]<t[k]) k=j;
        int tmp = t[k];
        t[k] = t[i];
        t[i] = tmp;
    }
}
```

Co to jest za algorytm?

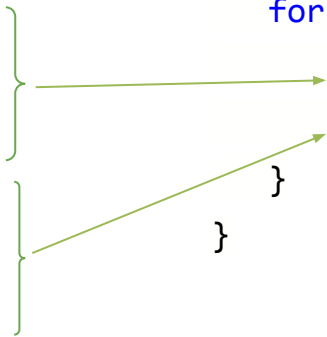
```
void algorithm1(auto from, auto to)
{
    for (; from != to; ++from) {
        auto iter = std::min_element(from, to);
        std::iter_swap(from, iter);
    }
}
```



Porównanie - selection sort

```
void selection_sort(int n, int t[])
{
    for(int i=0; i<n; i++) {
        int k=i;
        for(int j=i+1; j<n; j++) {
            if(t[j]<t[k]) k=j;
        }
        int tmp = t[k];
        t[k] = t[i];
        t[i] = tmp;
    }
}
```

```
void selection_sort(auto from, auto to)
{
    for (; from != to; ++from) {
        auto iter = std::min_element(from, to);
        std::iter_swap(from, iter);
    }
}
```

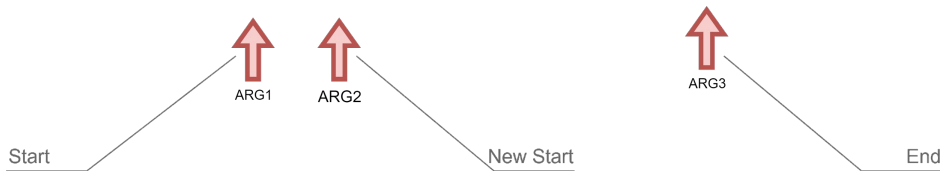
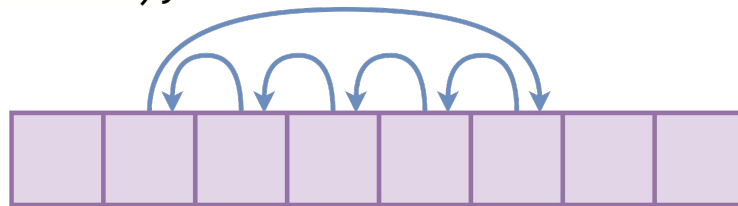


Co to jest za algorytm?

```
void algorithm2(int n, int arr[])
{
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Co to jest za algorytm?

```
void algorithm2(auto from, auto to)
{
    for (auto curr = from; curr != to; ++curr) {
        auto position = std::upper_bound(from, curr, *curr);
        std::rotate(position, curr, curr+1);
    }
}
```



Porównanie - insertion sort

```
void insertion_sort(int n, int arr[])
{
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```





```
void insertion_sort(auto from, auto to)
{
    for (auto curr = from; curr != to; ++curr) {
        auto position = std::upper_bound(from, curr, *curr);
        std::rotate(position, curr, curr+1);
    }
}
```

Quicksort?



Algorytmy

C++11

- all_of, any_of, none_of
- for_each_n 
- find_if_not
- copy_if
- copy_n
- move
- sample 
- is_partitioned
- partition_copy
- partition_point
- is_sorted
- is_sorted_until
- minmax
- minmax_element
- clamp 
- iota
- reduce, transform_reduce 

wybiera losowy element z zakresu

przycina wartość do zakresu

wypełnia zakres kolejnymi wartościami

- destroy_at 
- destroy 
- destroy_n 

wywołuje destruktor obiektu

All_of, any_of, none_of

```
bool all_positive = std::all_of(v.begin(), v.end(), [](int a) { return a > 0; });
```

- Sprawdza czy wszystkie, chociaż jedna czy żadna z wartości zakresu spełnia określony predykat
- Wykonanie kończy się jak tylko znany będzie wynik (np. po pierwszej wartości niespełniającej kryterium dla all_of)

Count, count_if

```
size_t cnt_fives = std::count(v.begin(), v.end(), 5);  
size_t cnt_pos = std::count_if(v.begin(), v.end(), [](int a) { return a > 0; });
```

- zlicza, ile elementów zakresu spełnia dany predykat
- złożoność liniowa

Find, find_if, find_if_not

```
T::iterator found = std::find(v.begin(), v.end(), 5);  
T::iterator found2 = std::find_if(v.begin(), v.end(), [](int a) { return a > 0; });  
T::iterator fnot = std::find_if_not(v.begin(), v.end(), [](int a) { return a <= 0; });
```

- znajdują pierwsze wystąpienie wartości lub wartość spełniającą predykat
- zwracają iterator na znaleziony element
- przy braku dopasowania zwracają koniec zakresu (iterator poza ostatni element)

Copy, copy_if, copy_n

```
std::copy(v.begin(), v.end(), dest.begin());  
std::copy_n(v.begin(), 5, dest.begin());  
std::copy_if(v.begin(), v.end(), dest.begin(), [](int a) { return a > 0; });
```

- kopują zakres elementów (bezwarunkowo lub wg predykatu) do innego iteratora
- dwa pierwsze parametry (iteratory lub rozmiar) określają zakres elementów do skopiowania
- trzeci parametr (iterator) określa miejsce docelowe
- użyteczne przy wykorzystaniu adapterów iteratorów
- jest też `std::move`

Remove, remove_if

```
T::iterator iter = std::remove(v.begin(), v.end(), 5);  
T::iterator iter = std::remove_if(v.begin(), v.end(), [](int a) { return a > 0; });
```

- Usuwają elementy z zakresu
- Zwracają iterator na pierwszy element poza nowym zakresem
- Nie usuwają (fizycznie) elementów z kolekcji
 - trzeba np. wywołać erase()

Transform, generate, iota, accumulate

```
std::transform(v.begin(), v.end(), strings.begin(), [](int a) { return std::to_string(a); });  
std::generate(v.begin(), v.end(), [val=0]() mutable { return val++; });  
std::iota(v.begin(), v.end(), 0);
```

- transform
 - Wykonuje jednoargumentowy operator na każdym elemencie zakresu i zapisuje wynik w iteratorze
- generate
 - Dla każdego elementu zakresu przypisuje wartość wynikającą z funkcji generatora
- iota
 - Wypełnia zakres elementów sekwencją wzrastających wartości począwszy od wartości ostatniego parametru wywołania

Transform, generate, iota, accumulate

```
size_t sum = std::accumulate(v.begin(), v.end(), 0u);  
size_t product = std::accumulate(v.begin(), v.end(), 1u, [](size_t result, int next) { return result*next; });
```

- Wykonuje operator+() na zakresie elementów i zbiera wynik agregując go z wartością ostatniego parametru
- Wykonuje podany operator dwuargumentowy na zakresie elementów i zbiera wynik poczynając od wartości ostatniego parametru

Random_shuffle, shuffle,

```
std::random_device rd;  
std::mt19937 gen(rd());  
std::shuffle(v.begin(), v.end(), gen);
```

- Wykonuje losową permutację elementów zakresu korzystając z podanego generatora losowości

Sort, lower_bound, upper_bound

```
std::sort(v.begin(), v.end());  
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });  
T::iterator iter = std::lower_bound(v.begin(), v.end(), 5);  
T::iterator iter = std::upper_bound(v.begin(), v.end(), 5);  
T::iterator iter = std::lower_bound(v.begin(), v.end(), 5, std::greater);  
T::iterator iter = std::upper_bound(v.begin(), v.end(), 5, std::greater);
```

- Sortuje zakres elementów poprzez operator<() lub podany operator dwuargumentowy
 - std::less, std::greater
- *_bound: zwracają iterator na element posortowanego zakresu, gdzie trzeba wstawić podaną wartość, żeby zakres dalej był posortowany
- lower_bound zwraca miejsce najbardziej z lewej strony
- upper_bound zwraca miejsce najbardziej z prawej strony



Totolotek (6 z 49)





SAPER

Wypełnij planszę do sapera minami





Algorytmy

Wylosuj 100 liczb. Sprawdź ile jest liczb parzystych (P) i policz sumę wszystkich liczb nieparzystych mniejszych od P.





Algorytmy

Wygeneruj 1000 liczb losowych. Stwórz wektor, który zawiera czynniki pierwsze każdej z wygenerowanych liczb (oraz samą liczbę). Posortuj wektor wg liczby czynników w kierunku malejącym (od większej do mniejszej liczby czynników). Wypisz pięć liczb o największej liczbie czynników.



Policz mniejsze od zera i nieparzyste

```
size_t count_odd_neg = std::count_if(i1, i2, [](auto v) { return v < 0 && v % 2 == 1; });

auto neg = [](auto v) { return v < 0; }; auto odd = [](auto v) { return v % 2 == 1; };
auto both = [](auto p1, auto p2) {
    return [p1, p2](auto v) { return p1(v) && p2(v); };
};
size_t c_o_n = std::count_if(i1, i2, both(neg, odd));

auto lessThanX = [](auto x) {
    return [x](auto v) { return v < x; };
};

std::count_if(i1, i2, both(lessThanX(5), odd));
```

Iterator

Operacyjne wzorce projektowe

- Zastosowania

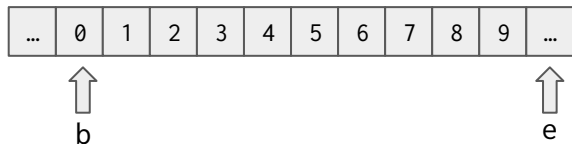
- dostęp do zawartości złożonego obiektu bez ujawniania jego wewnętrznej struktury
- możliwość jednoczesnego przechodzenia przez agregat przez kilka źródeł
- uproszczenie interfejsu agregatu
- możliwość wielokrotnej iteracji po agregacie
- możliwość zróżnicowania sposobu przejścia przez agregat (np. przeglądanie drzewa wszerz lub włąb) bez modyfikacji interfejsu agregatu

- Realizacja

- *kursor* - stan przejścia wyniesiony jest poza kontener
- *iterator zewnętrzny (aktywny)* - użytkownik kontroluje proces iteracji
 - `for(auto iter = std::begin(v); iter != std::end(v); std::advance(iter, 1)) { ... }`
 - `for(auto item: v) { ... }`
- *iterator wewnętrzny (pasywny)* - iterator kontroluje proces iteracji i wywołuje zleconą funkcję
 - `std::for_each(std::begin(v), std::end(v), [](auto item) { ... });`

```
int tablica[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* b = tablica; // &tablica[0]
int* e = tablica+10; // &tablica[10]

std::for_each(b, e, [](int v) {
    std::cout << v << std::endl;
});
```



```
template<typename T> class Tab {
public:
    const T* begin() const { return m_data; }
    const T* end() const { return m_data+42; }
private:
    T m_data[42];
};
```

```
Tab<double> t;
// ...
for(const double v: t) { ... }
```

Rodzaje iteratorów

- InputIterator

- ForwardIterator

- `reference operator*() const`
 - `pointer operator->() const`
 - `iterator& operator++()`
 - `iterator operator++(int)`
 - `operator==(iterator) const`
 - `operator!=(iterator) const`

- BidirectionalIterator

- `iterator& operator--()`
 - `iterator operator--(int)`

- RandomAccessIterator

- swobodny dostęp

- ContiguousIterator (C++20)

- OutputIterator

- pozwala na zapis do wskazywanego obiektu

Adaptery iteratorów

- `reverse_iterator`
 - `make_reverse_iterator`
- `move_iterator`
 - `make_move_iterator`
- `back_insert_iterator`
 - `back_inserter`
- `front_insert_iterator`
 - `front_inserter`
- `insert_iterator`
 - `inserter`

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

```
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    std::vector<int> odd;

    std::copy_if(v.begin(), v.end(), std::back_inserter(odd), [](int e) { return e % 2; });

    std::copy(odd.begin(), odd.end(), std::ostream_iterator<int>(std::cout, " ")); std::cout << '\n';

    return 0;
}
```

zakres

iterator wyjściowy

predykat

Funkcje związane z iteratorem

`std::begin()`

`std::end()`

`std::advance()`

`std::distance()`

Implementacja iteratora

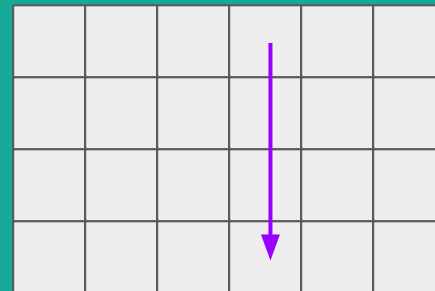
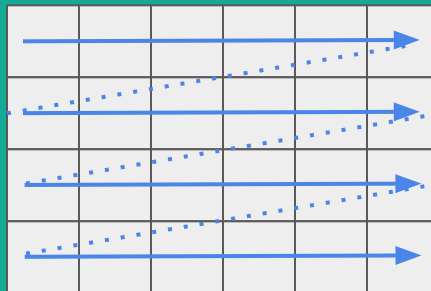
- `std::iterator` - pomocnicza klasa szablonowa
 - `using`
 - `iterator_category`
 - `input_iterator_tag`
 - `output_iterator_tag`
 - `forward_iterator_tag`
 - `bidirectional_iterator_tag`
 - `random_access_iterator_tag`
 - `contiguous_iterator_tag` (C++20)
 - `value_type`
 - `difference_type`
 - `pointer`
 - `reference`
 - odpowiednie metody
- `ew. std::begin(), std::end(), itd.`
- `std::iterator_traits`



Iterator

Zaimplementuj iteratory dla Matrix

- przechodzący przez wszystkie komórki
 - input iterator
 - bidirectional iterator
 - random access iterator
- przechodzący przez określoną kolumnę



Zakresy



Jak przeiterować przez tablicę elementów?

```
int tab[10];
```

```
for(size_t i = 0; i < 10; ++i) {  
    use(tab[i]);  
}
```

```
for(Ptr iter = tab; iter < tab+10; ++iter) {  
    use(*iter);  
}
```

Zakresowe for



```
for ( deklaracja : wyrażenie) { }
```

równoważne:

```
{  
    auto __expr = wyrażenie;  
    for (auto __start = begin(__expr), __end = end(__expr); __start != __end; ++__start) {  
        deklaracja = *__start;  
        { }  
    }  
}
```

Zakresowe for

- Lukier składniowy na pętlę for
- Deklaracja zmiennej często jest typu auto
- Deklaracja powinna mieć kwalifikator const
- Nie wolno modyfikować kontenera, po którym się chodzi
 - można modyfikować zawartość elementów

Przejdźcie po jednym słowie ze stringa

```
std::string_view text;  
for (auto iter = text.begin(); iter != text.end() && *iter != ' '; ++iter)  
{  
    putchar(*iter);  
}
```

```
struct WordView {
    std::string_view m_v;
    WordView(std::string_view v) : m_v(v) {}
    const char *begin() const { return std::data(m_v); }
    const char *end() const {
        // ???
    }
};
```

Zakresowe for w C++17



```
for (auto from = c.begin(), auto to = c.end(); from != to; ++from) {  
    auto&& elem = *from;  
    body(elem);  
}
```

// typy “from” i “to” mogą się różnić!

for ze strażnikiem

```
struct WordView {
    std::string_view m_v;
    WordView(std::string_view v) : m_v(v) {}
    const char *begin() const { return std::data(m_v); }
    struct sentinel_t {};
    sentinel_t end() const { return {}; }
    friend bool operator!=(const char *v, WordView::sentinel_t) {
        return *v != '\0' && *v != ' '; // simplification! sv may not be \0 terminated!
    }
};
```

Zakres

- para iterator + strażnik
- iterator jest bardzo dobrym strażnikiem

Algorytmy

```
std::vector v = {10, 15, -4, 8, 0, 2, 8, 1, 21};  
std::sort(v.begin(), v.end());
```

Algorytmy działające na zakresach

```
#include <algorithm>
```

```
std::vector v = {10, 15, -4, 8, 0, 2, 8, 1, 21};  
std::ranges::sort(v);
```

```
struct Person {  
    enum class Gender { Female, Male };  
    string name;  
    Gender gender{Gender::Female};  
    unsigned short age{};  
};
```

Znajdź liczbę Piotrów

```
auto nameIsPiotr = [](const auto &e) { return e.name == "Piotr"; };  
const auto cnt = std::count_if(v.begin(), v.end(), nameIsPiotr);
```

```
const auto cnt = std::ranges::count_if(v, nameIsPiotr);
```

Odwzorowania (projections)

`{ile} count({zakres}, {wartość}, {odwzorowanie})`

```
auto projection = [](const auto &e) { return e.name; };  
const auto cnt = std::ranges::count(v, "Piotr", projection);
```

Odwzorowania (projections)

```
const auto cnt = std::ranges::count(v, "Piotr", &Person::name);
```

Odwzorowanie może być czymkolwiek, co da się wywołać przez `std::invoke()`

- funkcja/obiekt funkcyjny
- wskaźnik na metodę klasy
- wskaźnik na pole klasy

Ekspresja

```
auto nameIsPiotr = [](const auto &e) { return e.name == "Piotr"; };  
const auto cnt = std::ranges::count_if(v, nameIsPiotr);
```

```
const auto cnt = std::ranges::count(v, "Piotr", &Person::name);
```

Posortuj pojemnik wg wieku

```
auto lessThanByAge = [](const auto &e1, const auto &e2) {  
    return e1.age < e2.age;  
};
```

```
std::sort(v.begin(), v.end(), lessThanByAge);
```


Posortuj pojemnik wg wieku

```
auto lessThanByAge = [](const auto &e1, const auto &e2) {  
    return e1.age < e2.age;  
};
```

```
std::ranges::sort(v, lessThanByAge);
```

Posortuj pojemnik wg wieku (odwzorowanie)

```
std::ranges::sort(v, std::less<>{}, &Person::age);
```

Znajdź kobietę

```
auto isFemale = [](const auto &e) {  
    return e.gender == Person::Gender::Female;  
};  
  
auto iter = std::find_if(v.begin(), v.end(), isFemale);
```

Znajdź kobietę

```
auto iter = std::ranges::find(v, Person::Gender::Female, &Person::gender);
```

- odwzorowanie zwraca płeć
- algorytm zwraca element właściwego pojemnika (Person)

Policz sumę wieku wszystkich osób

RANGE-V3

```
auto sum = ranges::accumulate(v, 0u, std::plus<>{}, &Person::age);
```

Policz sumę wieku kobiet

```
std::vector<Person> women;  
ranges::copy_if(v,  
                std::back_inserter(women),  
                [](auto g) { return g == Person::Gender::Female; },  
                &Person::gender);  
auto sum = ranges::accumulate(women, 0u, std::plus<>{}, &Person::age);
```

Problemy z algorytmami STL

- algorytmy słabo się komponują
 - po każdym kroku trzeba robić kolejny pojemnik z wynikami pośrednich obliczeń

Adaptery

```
struct Person {  
    enum class Gender { Female, Male };  
    string name;  
    Gender gender{Gender::Female};  
    unsigned short age{};  
};
```

```
auto isFemale = [](const auto &p) {  
    return p.gender == Person::Gender::Female;  
};
```

```
auto women = ranges::views::filter(v, isFemale);  
auto sum = ranges::accumulate(women, 0u, std::plus<>{}, &Person::age);
```

RANGE-V3

Adaptery

```
struct Person {  
    enum class Gender { Female, Male };  
    string name;  
    Gender gender{Gender::Female};  
    unsigned short age{};  
};
```

```
auto isFemale = [](const auto &p) {  
    return p.gender == Person::Gender::Female;  
};
```

```
auto women = std::ranges::views::filter(v, isFemale);  
const auto sum = std::accumulate(women.begin(), women.end(), 0u,  
    [](auto result, const auto &p) { return result + p.age; });
```

STD::RANGES

Widoki

- fabryki
 - tworzą nowe widoki
- adaptery
 - dostosowują istniejące widoki

Totolotek

```
std::mt19937 rd{std::random_device{}}();  
std::array<int, 6> out;  
auto in = std::ranges::views::iota(1, 50);  
std::ranges::sample(in, out.begin(), 6, rd);
```

Leniwość widoków

- widoki wykonują pracę dopiero wtedy, gdy ktoś poprosi o kolejny element
 - `iota(x)` generuje kolejne wartości od `x` do nieskończoności
 - nie ma to nic wspólnego ze współprocedurami (coroutines)
- dwa sposoby użycia adapteru
 - nałożyć na istniejący widok
 - `auto women = std::ranges::views::filter(all_people, isFemale);`
 - traktować jako funkcję modyfikującą dowolny widok
 - `auto woman_filter = std::ranges::views::filter(isFemale);`

Suma lat kobiet do emerytury

```
namespace rv = std::ranges::views;  
  
auto women = rv::filter(isFemale);  
auto until_retirement = rv::transform([](const auto &p) { return 60-p.age; });  
auto w_u_r = until_retirement(women(v));  
auto sum = std::accumulate(  
    std::ranges::begin(w_u_r),  
    std::ranges::end(w_u_r),  
    0u  
); // co z kobietami starszymi niż 60?
```

Suma lat kobiet do emerytury

```
namespace rv = std::ranges::views;  
  
auto women = rv::filter(isFemale);  
auto until_retirement = rv::transform([](const auto &p) { return 60-p.age; });  
auto before_retirement = rv::filter([](const auto &p) { return p.age < 60; });  
  
auto w_u_r = until_retirement(before_retirement(women(v)));  
  
auto sum = ranges::accumulate(w_u_r, 0u); // range-v3  
auto cnt = std::ranges::distance(w_u_r);  
auto avg = cnt ? sum * 1.0 / cnt : 0.;
```

Składanie widoków

`last(middle(first(v)))` \rightarrow `v | first | middle | last`

Składanie widoków

```
using namespace std::ranges::views;
```

```
auto is_woman = filter([](const auto& p) { return p.gender == Person::Gender::Female; });
```

```
auto is_before_retirement = filter([](const auto& p) { return p.age < 60; });
```

```
auto years_until_retirement = transform([](const auto& p) { return 60 - p.age; });
```

```
auto sum = ranges::accumulate(v|is_woman|is_before_retirement|years_until_retirement, 0u);
```

RANGE-V3

```
auto final_view = v
```

```
    | is_woman
```

```
    | is_before_retirement
```

```
    | years_until_retirement;
```

```
auto sum = std::accumulate(final_view.begin(), final_view.end(), 0u);
```

STD::RANGES

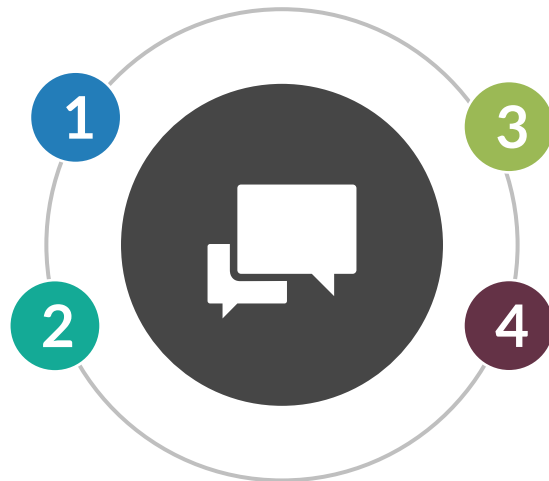
Plan szkolenia

Podstawy

Przegląd mechanizmów wprowadzonych w nowych standardach C++

Algorytmy i iteratory

Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



Szablony, metaprogramowanie

Programowanie na szablonych,
programowanie w czasie kompilacji

Pytania i podsumowanie

Wyjaśnienie nurtujących kwestii, odpowiedzi
na pytania

Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

C radzi sobie poprzez makro:

```
#define min(a, b) (a < b ? a : b)
```

Problem:

```
m = min(fun(42), fun(58));
```

fun() o niższym wyniku wywoła się dwa razy!

Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

Zdefiniujmy funkcję!

```
int min(int a, int b) { return a < b ? a : b; }
```

Co z wcześniejszym problemem?

```
auto a = min(fun(42), fun(58)); // fun() o niższym wyniku wywoła się raz
```

Sukces! Czy nie?

```
auto a = min(4.5, 5.7);  
std::cout << a << endl; // wynik to '4'
```

min() rzutuje argumenty na int! co więcej, zwraca int!

Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

- Należy zdefiniować rodzinę funkcji (przeciążenia)
 - `int min(int a, int b)` { `return a < b ? a : b;` }
 - `float min(float a, float b)` { `return a < b ? a : b;` }
 - `double min(double a, double b)` { `return a < b ? a : b;` }
 - `uint64_t min(uint64_t a, uint64_t b)` { `return a < b ? a : b;` }
- Kompilator dobiera odpowiednie przeciążenie na podstawie parametrów.
- Jeżeli przeciążenie nie istnieje, zwróci błąd
 - np. podaliśmy dwa niekompatybilne typy do `min()`

Szablony

Szablony funkcji

- wzorzec klasy, funkcji lub zmiennej, który może zostać wykorzystany przez kompilator do utworzenia prawdziwej klasy, funkcji lub zmiennej
- parametryzowany typami, wartościami lub szablonami

```
template<typename T>
```

```
const T& min(const T& a, const T &b) { return a < b ? a : b; }
```

Szablony funkcji

Szablony funkcji

- szablon funkcji to nie funkcja tylko ogólny opis algorytmu
 - narzędzie do generowania funkcji
 - deklaracja szablonu bierze udział w rozwiązywaniu przeciążeń funkcji
 - jeżeli kompilator wybierze szablon, będzie potrzebował jego ciała, żeby wyprodukować funkcję
-
- lista parametrów szablonu funkcji
 - lista parametrów szablonu
 - lista parametrów funkcji
 - T jest zastępnikiem jakiegoś typu, który zostanie określony przez kompilator podczas kompilacji

```
template <typename T>
const T& min(const T &a, const T &b) {
    // ...
}
```

lista parametrów szablonu

lista parametrów funkcji

Szablony funkcji

Instancjacja szablonu

- generacja definicji funkcji z szablonu nazywa się **instancjacją szablonu**
- definicja funkcji wygenerowana z szablonu czasem nazywa się **instancjacją** (rzeczownik)
- instancjowanie szablonów jest częścią procesu kompilacji (odbywa się w dwóch krokach)

- wywołanie funkcji polega na wywołaniu po nazwie konkretnej instancjacji

```
int i, j;
```

```
auto m = min<int>(i, j);
```

- zmusza to kompilator do zinstancjonowania definicji szablonu

```
const int& min<int>(const int &a, const int &b);
```

- w tym przypadku za “T” kompilator podstawia “int”

Szablony funkcji

- szablon może mieć dowolną liczbę parametrów szablonych

```
template<typename U, typename V>
```

```
V foo(U u, V v, int a) { ... }
```

- wywołanie też będzie zawierać te parametry

```
int x = foo<K1, K2>(i1, i2, 7);
```

- zamiast “typename” można używać “class”

- niezależnie czy to, co podajemy, jest klasą czy typem wbudowanym
- jest jedna sytuacja (przed C++17), gdzie tych słów nie można używać zamiennie



szablony funkcji

- suma elementów w wektorze
- funkcja pobierająca tablicę wartości i wywołująca na każdej podaną funkcję



Szablony klas

- szablon klasy to nie typ danych tylko generalizacja typu
- przepis, którego kompilator może użyć żeby tworzyć podobne (niezależne) klasy

```
template<typename T>
class rational {
public:
    rational();
    rational(T n);
    rational(T n, T d);
    // ...
private:
    T num, den;
};
```

Szablony klas

Użycie szablonu

```
rational<int> r1{10, 2};  
using rat_int = rational<int>;  
rat_int r2{15};
```

Składowe funkcje szablonów klasy

- ```
template<typename T>
class rational {
public:
 // ...
 rational(T n);
 // ...
};
```

definicja może być poza  
deklaracją klasy

```
template<typename T>
rational<T>::rational(T n) { ... }
```

**nazwa klasy**

**nazwa funkcji**

# Parametry szablonów

---

- argumentem szablonu są nie tylko typy, ale i wyrażenia

```
template<size_t N> class IntArray { ... };
```

- parametr nie będący typem może być:
  - liczbą całkowitą
  - wyliczeniem
  - wskaźnikiem
  - wskaźnikiem do składowej klasy
  - referencją
  - nullptr
  - liczbą rzeczywistą (C++20)
  - obiektem klasy literału (C++20)

# Dedukcja typów argumentów szablonów

Szablony funkcji

```
i = min(a, b);
```

- kompilator sprawdzi typy argumentów wywołania funkcji (ale nie zwracany typ)
- na tej podstawie “uzupełni” brakujące parametry szablonów

```
template <typename T> T min(T a, T b);
```

- `min(10, 12);` // wywoła `min<int>(10, 12)`
- `min(10.1, 12.2);` // wywoła `min<double>(10.1, 12.2)`
- `min(10, 10.1);` // błąd kompilacji -> różne typy
- `min<int>(10, 10.1);` // wywoła `min<int>(10, 10.1)`

# Dwie fazy kompilacji szablonów

---

- kompilator zbiera dostępne deklaracje szablonów
  - nie generuje kodu
  - nie interesuje go, jakie są parametry szablonów
  - sprawdza poprawność składni tej części ciała, która nie zależy od parametrów szablonu
- kompilator instancjonuje potrzebny mu szablon dla konkretnej kombinacji parametrów szablonowych
  - sprawdza poprawność składni ciała szablonu, dla konkretnych parametrów szablonowych

# Typy zależne (typename)

---

```
template<typename T>
T::size_type process(const T& arg) {
 T::size_type * iter (T::len);
}
```

- funkcja działa tylko dla typu T, który zawiera składowe size\_type oraz len
- czym jest T::size\_type?
- czym jest T::len?
- czym jest iter?
- czym jest \*?



# Typy zależne (typename)

---

```
template<typename T>
typename T::size_type process(const T& arg) {
 typename T::size_type * iter (T::len);
}
```

- w razie wątpliwości standard nie pozwala kompilatorowi założyć, że coś jest typem
- musimy mu to wskazać
- często kompilator wie, co mieliśmy na myśli i sugeruje rozwiązanie
- w nowszych standardach niektóre typename nie są już potrzebne

# Specjalizacja szablonu

---

- podstawienie pewnej konkretnej kombinacji argumentów za parametry szablonu
- domniemana
  - kiedy kompilator dopasowuje argumenty do szablonu
- jawna
  - kiedy programista wymusza konkretną implementację funkcji lub klasy dla danego zestawu argumentów szablonowych

# Specjalizacje szablonu

---

```
auto x = min<const char *>("def", "abc");
```

```
const char * min(const char *a, const char *b) { return a < b ? a : b; }
```

funkcja porówna wskaźniki

```
template<>
const char * min(const char *a, const char *b) {
 return strcmp(a, b) < 0 ? a : b;
}
```

# Specjalizacje szablonu

---

- częściowa
  - część parametrów szablonowych jest zastępowana konkretnymi argumentami
- pełna
  - wszystkie parametry szablonowe są zastępowane konkretnymi argumentami

# constexpr



# Wyrażenia stałe (constexpr)

---

C++11

- określa, że możliwe jest obliczenie wartości funkcji lub zmiennej w czasie kompilacji
- pozwala na użycie ich w kontekście, gdzie podaje się stałe wartości, które muszą być znane podczas kompilacji
  - rozmiary tablic
  - wartości w switch-case
  - parametry szablonów
- implikuje, że funkcja zwraca zawsze tę samą wartość dla tych samych argumentów

# Zmienne constexpr

---

C++11

- literał
  - skalar, referencja lub tablica
  - trywialny destruktork
  - agregacja lub przynajmniej jeden konstruktor constexpr
- koniecznie inicjowana podczas deklaracji
- inicjalizacja wyłącznie przy pomocy wyrażeń stałych
  - włączając w to ewentualne konwersje
- używane najczęściej w ramach funkcji constexpr

```
constexpr int magic_number = std::min(7, 8);
```

# Funkcje constexpr

---

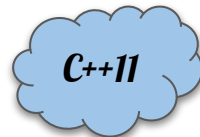
A blue cloud-shaped logo with the text "C++11" inside.

- niewirtualna
- zwracany typ jest literałem (lub void)
- wszystkie parametry są literałami
- musi istnieć przynajmniej jedna ścieżka wykonania, która może zostać obliczona podczas kompilacji
- automatycznie definiuje funkcję jako inline



# C++11

---



- co jest dozwolone?
  - skasowanie funkcji (=delete)
  - static\_assert, typedef i podobne, które nic nie “wykonują”
  - wyłącznie pojedyncze wyrażenie return
  - wołanie funkcji i odczyt zmiennych constexpr
- co jest zabronione?
  - wszystko co nie jest dozwolone :)
    - m.in. zmienne, pętle, switch

```
constexpr unsigned long factorial(unsigned n) {
 return n <= 1 ? 1 : n * factorial(n-1);
}
```

```
constexpr unsigned long f10 = factorial(10);
```



## Liczby pierwsze

---

Zaimplementuj funkcję `constexpr`, która sprawdzi, czy dana liczba jest pierwsza.



# C++14

---



- co jest dozwolone?
  - wszystko co nie jest zabronione
- co jest zabronione?
  - bloki asm
  - goto
  - etykiety (wyłączając switch-case)
  - wyjątki
  - dynamiczna alokacja pamięci
  - definiowanie zmiennych nie będących literałami
  - definiowanie zmiennych statycznych lub thread local
  - definiowanie zmiennych bez ich inicjalizacji

# Silnia constexpr - iteracyjnie

---

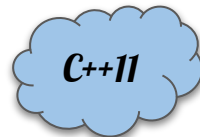
C++14

```
constexpr unsigned long factorial_i(unsigned n) {
 unsigned long result = 1;
 for (unsigned a = 1; a < n; ++a) result *= a;
 return result;
}
```

```
constexpr unsigned long f10 = factorial_i(10);
```

# Konstruktory constexpr

---



- parametry muszą być obliczalne podczas kompilacji
- klasa nie może mieć wirtualnej klasy bazowej
- konstruktor nie używa wyjątków

# C++14

---



- co jest dozwolone?
  - skasowanie (=delete)
  - domyślny (=default)
  - static\_assert, typedef i podobne
  - wyrażenia dozwolone dla funkcji constexpr
  - każda składowa musi zostać zainicjowana
  - wywoływane mogą być tylko konstruktory constexpr

# Czy to jest prawidłowe?

---

```
constexpr int funkcja(int i) {
 return i;
}
```



# Czy to jest prawidłowe?

---

```
constexpr int funkcja(int i) {
 if (i < 0) {
 std::cout << "Liczba jest ujemna" << std::endl; // !!!!
 }
 return i;
}
```

# UWAGA!

`constexpr`

---

Oznaczenie funkcji jako `constexpr` wcale nie oznacza, że ona wykona się w trakcie kompilacji!

Oznaczenie funkcji jako `constexpr` wcale nie oznacza, że może się ona wykonać w trakcie kompilacji dla dowolnego (prawidłowego) zestawu argumentów!



switch-constexpr

---

fnn1a





Date constexpr

---



# Zmienne szablonowe

---



- pozwalają na generyczne definicje zmiennych w zależności od podanych parametrów

```
template<typename T> constexpr T Pi = T(3.14159265L);
std::cout << "Integer value of Pi is: " << Pi<int> << std::endl;
```

# Zmienne szablonowe - przykład

---

C++14

```
using stringvector = std::vector<std::string>;
template<size_t L> std::string MyName = std::string("John Smith").substr(0, L);
template<> std::string MyName<0> = "???";
template<> std::string MyName<2> = "JS";
template<> std::string MyName<4> = "J.S.";
template<> std::string MyName<5> = "J. S.";
template<> std::string MyName<6> = "JSmith";
template<> std::string MyName<7> = "John S.";

int main() {
 stringvector v = expand(std::make_index_sequence<10>());
 for (const std::string &elem : v) {
 std::cout << "My name is " << elem << "." << std::endl;
 }
 return 0;
}
```

# Silnia - na zmiennych

---



```
template<int V> const unsigned int factorial = V*factorial<V-1>;
template<> const unsigned int factorial<1> = 1;
template<> const unsigned int factorial<0> = 1;
```

# Zmienne szablonowe

Zastosowanie

---

```
template<unsigned N>
unsigned long long factorial = N*factorial<N-1>;
```

```
template<>
unsigned long long factorial<1> = 1;
```



# If constexpr

---

A green cloud-shaped logo with the text "C++17" inside.

```
template<typename T> void print_type(T v) {
 if constexpr (std::is_pointer_v<T>) {
 std::cout << "Is a pointer" << std::endl;
 } else {
 std::cout << "Not a pointer" << std::endl;
 }
}
```

# Metaprogramowanie

---

**Metaprogramowanie** to pisanie programów komputerowych...

1. Które piszą lub manipulują innymi programami (lub nimi samymi) jakby były ich danymi, lub
2. Które robią podczas kompilacji coś, co zwykle robi się podczas działania programu

**Metaprogramowanie szablonowe** w C++ wykorzystuje **instancję szablonów** do wykonania kodu w trakcie kompilacji:

- Użycie nazwy szablonu w miejsce funkcji, typu lub zmiennej powoduje, że kompilator **powoła** oczekiwaną encję na podstawie tego szablonu
- Metaprogramowanie szablonowe wykorzystuje ten mechanizm w celu poprawy **uniwersalności kodu i prędkości jego wykonania**.

źr. Walter E. Brown

# IntIdentity

---

```
template<int N>
struct Identity {
 static constexpr int value = N;
};
```

# Identity

---

```
template<typename T, T N>
struct Identity {
 static constexpr T value = N;
};
```

# Suma

---

```
template<typename T, T v1, T v2>
struct Sum {
 static constexpr T value = Identity<T, v1+v2>::value;
};
```

```
template<typename T, T v1, T v2>
struct Sum : Identity<T, v1+v2>{};
```



Silnia na identity

---



---

```
template<int N> struct abs {
 static constexpr int value = N < 0 ? -N : N;
};
```

```
template<int N> constexpr int abs_v = abs<N>::value;
```

# Metafunkcje vs constexpr

---

```
constexpr int abs(int N) { return N < 0 ? -N : N; }
```

Różnice:

- Szablony generują struktury, które mogą mieć wiele “składowych”:
  - typy
  - stałe
  - deklaracje metod
  - definicje metod constexpr



---

```
template<int N> struct factorial {
 static_assert(N>1);
 static constexpr int value = N * factorial<N-1>::value;
};
```

```
template<> struct factorial<1> {
 static constexpr int value = 1;
};
```

```
template<> struct factorial<0> {
 static constexpr int value = 1;
};
```

```
int main() {
 std::cout << factorial<5>::value << std::endl;
 return 0;
}
```



- Metafunkcje mogą otrzymać typ jako swój argument, funkcje `constexpr` nie mają takiej możliwości.
- Metafunkcje mogą zwrócić typ, funkcje `constexpr` nie

# Cechy typów

---

Interfejs do sprawdzania/odpytywania o właściwości typów  
działający podczas kompilacji

---

```
template<typename T, T v>
struct integral_constant { static constexpr T value = v; };
```

```
template<bool B>
using bool_constant = integral_constant<bool, B>;
```

```
using false_type = bool_constant<false>;
using true_type = bool_constant<true>;
```

# is\_void, is\_reference

---

```
template<typename T> struct is_void : false_type {};
template<> struct is_void<void> : true_type {};
```

```
template<typename T> struct is_reference : false_type {};
template<typename U> struct is_reference<U&> : true_type {};
template<typename V> struct is_reference<V&&> : true_type {};
template<typename T> inline constexpr bool is_reference_v = is_reference<T>::value;
```



is\_vector\_or\_set

—



# remove\_const

---

```
template<typename T>
struct remove_const { using type = T; }
```

```
template<typename T>
struct remove_const<T const> { using type = T; }
```

```
template<typename T>
using remove_const_t = typename remove_const<T>::type;
```

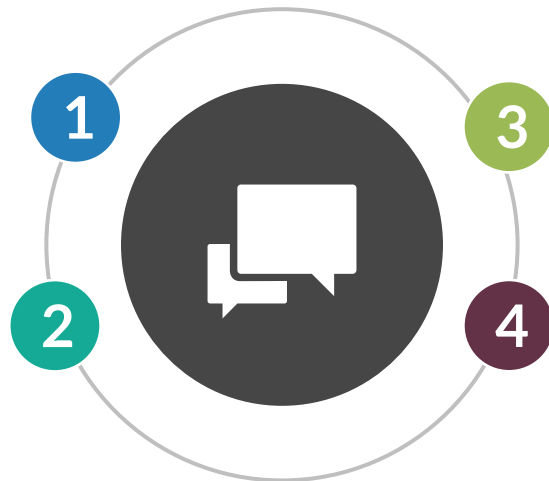
```
static_assert(std::is_same_v<remove_const_t<const int>, int>);
```

# Plan szkolenia

---

**Podstawy**  
Przegląd mechanizmów wprowadzonych w nowych standardach C++

**Algorytmy i iteratory**  
Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



**Szablony, metaprogramowanie**

Programowanie na szablonych,  
programowanie w czasie kompilacji

**Pytania i podsumowanie**

Wyjaśnienie nurtujących kwestii, odpowiedzi  
na pytania



# Plan szkolenia

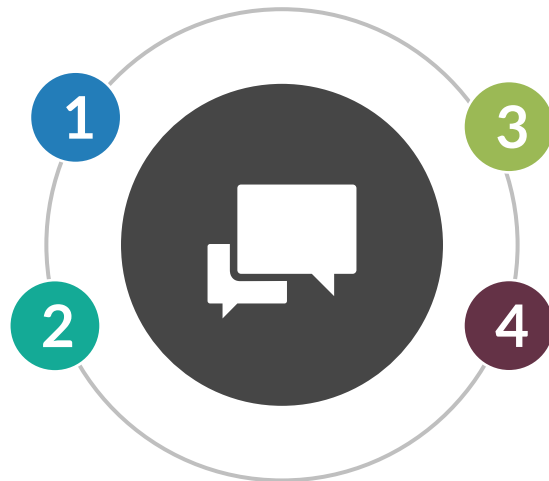
---

## Podstawy

Przegląd mechanizmów wprowadzonych w nowych standardach C++

## Algorytmy i iteratory

Programowanie uogólnione przy użyciu rozwiązań z biblioteki standardowej



## Szablony, metaprogramowanie

Programowanie na szablonych,  
programowanie w czasie kompilacji

## Pytania i podsumowanie

Wyjaśnienie nurtujących kwestii, odpowiedzi  
na pytania



Dziękuję za uwagę

---

...

Kontakt:



[www.sages.pl](http://www.sages.pl)



[w.wysota@sages.io](mailto:w.wysota@sages.io)