

# Programowanie w języku C++17

Krystian Piękoś

# Informacje organizacyjne

- Czas trwania szkolenia:
  - 9:00 - 16:00
- Materiały szkoleniowe:
  - <https://infotraining.bitbucket.io/cpp-17>
  - repozytorium GIT
- Przerwy
- Ankieta

# Historia standaryzacji C++

- 1998 - pierwszy standard ISO C++ - C++98
- 2003 - C++03 - TC1 ("Technical Corrigendum 1")
- 2005 - opublikowany "Technical Report 1"
- 2011 - C++0x ratyfikowany jako C++11
- 2014 - C++14 - bugfixes & minor revision
- 2017 - C++17 - umiarkowana modyfikacja standardu
- 2020 - C++20 - duża modyfikacja standardu

# Stałe definiujące wersje

Stała preprocessora `_cplusplus` definiuje, która wersja standardu jest dostępna w trakcie kompilacji

Wersja standardu	Wartość stałej
C++98 & C++03	199711L
C++11	201103L
C++14	201402L
C++17	201703L

# C++17 w kompilatorze gcc

- Należy użyć opcji komplikacji:
  - `-std=c++17`

# Structured bindings

# Structured bindings

- umożliwiają zdefiniowanie i zainicjowanie wielu zmiennych w jednym wyrażeniu
- dedukcja typu odbywa się przy pomocy mechanizmu auto

# Structured bindings

```
01 template <typename Container>
02 auto calculate_stats(const Container& data)
03 {
04     auto [min_pos, max_pos] = minmax_element(begin(data), end(data));
05     double avg = accumulate(begin(data), end(data), 0.0) / size(data);
06
07     return std::tuple(*min_pos, *max_pos, avg);
08 }
09
10 std::vector data = {1, 42, 665, -2};
11
12 const auto [min, max, avg] = calculate_stats(data);
```

# Structured bindings

```
01 template <typename Container>
02 auto calculate_stats(const Container& data)
03 {
04     auto [min_pos, max_pos] = minmax_element(begin(data), end(data));
05     double avg = accumulate(begin(data), end(data), 0.0) / size(data);
06
07     return std::tuple(*min_pos, *max_pos, avg);
08 }
09
10 std::vector data = {1, 42, 665, -2};
11
12 const auto [min, max, avg] = calculate_stats(data);
```

# Structured bindings

```
01 template <typename Container>
02 auto calculate_stats(const Container& data)
03 {
04     auto [min_pos, max_pos] = minmax_element(begin(data), end(data));
05     double avg = accumulate(begin(data), end(data), 0.0) / size(data);
06
07     return std::tuple(*min_pos, *max_pos, avg);
08 }
09
10 std::vector data = {1, 42, 665, -2};
11
12 const auto [min, max, avg] = calculate_stats(data);
```

# Structured bindings - tablice

```
01 auto foo() → int(&)[2];  
02  
03 auto [first, second] = foo();
```

# Structured bindings - std::pair

```
01 std::set<int> unique_numbers = get_numbers();
02
03 if (auto [pos, is_inserted] = unique_numbers.insert(1); is_inserted)
04     std::cout << (*pos) << " has been inserted\n";
```

# Structured bindings - std::array

```
01 std::array<int, 4> get_data();
02
03 auto [i, j, k, l] = get_data();
```

# Structure bindings - klasy, struktury

- wszystkie niestatyczne składowe obiektu klasy/struktury/unii muszą być publiczne i być bezpośrednio zdefiniowane w klasie/strukturze wiązanego obiektu lub w jego klasie bazowej
  - anonimowe unie nie są dozwolone

```
01 struct Data
02 {
03     int n;
04     char c;
05     double d;
06 };
07
08 Data data1 { 1, 'A', 3.14 };
09
10 auto [member1, member2, member3] = data1;
```

## Mechanizm wiązania

Structured bindings wykorzystuje nową (anonimową) zmienną, a nowe identyfikatory wprowadzone w wiązaniu odwołują się do pól tej anonimowej zmiennej

# Mechanizm wiązania

```
01 struct Timestamp  
02 {  
03     int hours, minutes, seconds;  
04 };  
05  
06 Timestamp timestamp{12, 0, 30};  
07  
08 auto [h, m, s] = timestamp;
```

# Mechanizm wiązania

```
01 struct Timestamp  
02 {  
03     int hours, minutes, seconds;  
04 };  
05  
06 Timestamp timestamp{12, 0, 30};  
07  
08 auto [h, m, s] = timestamp;
```

odpowiada koncepcyjnie

# Mechanizm wiązania

```
01 struct Timestamp  
02 {  
03     int hours, minutes, seconds;  
04 };  
05  
06 Timestamp timestamp{12, 0, 30};  
07  
08 auto [h, m, s] = timestamp;
```

odpowiada koncepcyjnie

```
01 auto entity = timestamp;  
02 auto& h = entity.hours;  
03 auto& m = entity.minutes;  
04 auto& s = entity.seconds;
```

# Kwalifikatory dla wiązań

- Deklaracje *structured bindings* mogą być dekorowane kwalifikatorami w postaci:
  - referencji: & i &&
  - modyfikatorów const oraz volatile
  - deklaracji wyrównania alignas

# Kwalifikatory dla wiązań

- Deklaracje *structured bindings* mogą być dekorowane kwalifikatorami w postaci:
  - referencji: & i &&
  - modyfikatorów const oraz volatile
  - deklaracji wyrównania alignas

dekoracja taka dotyczy całego anonimowego obiektu!

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

x i y są odwołaniami do elementów encji, która jest kopią tablicy a

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

rx i ry są odwołaniami do elementów tablicy a

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

cx i cy są typu const int i są elementami encji, będącej kopią tablicy

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

id i pi odnoszą się do anonimowej encji, która jest wyrównana do 128 bajtów

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

ERROR - cannot bind auto& to rvalue std::tuple

# Kwalifikatory dla wiązań - kod

```
01 int a[] = { 42, 13 };
02
03 auto [x, y] = a;
04 auto& [rx, ry] = a;
05 const auto [cx, cy] = a;
06 alignas(128) auto[id, pi] = std::tuple(42, 3.14);
07 auto& [id, name] = std::make_tuple(1, "John"s);
08 auto&& [id, name] = std::make_tuple(1, "John"s);
```

OK!

## Semantyka przenoszenia

Aby przenieść obiekt do anonimowej zmiennej, należy użyć następującej konstrukcji:

```
01 Timestamp timestamp{12, 0, 30};  
02  
03 auto [h, m, s] = std::move(timestamp);
```

## Semantyka przenoszenia

W przypadku, kiedy użyjemy specyfikatora `auto&&`, obiekt `timestamp` wciąż przechowuje dane, ponieważ encja jest referencją do rvalue:

```
01 Timestamp timestamp{12, 40, 0};  
02  
03 auto&& [h, m, s] = std::move(timestamp);
```

# Structured bindings - iteracja po mapach

```
01 std::map<std::string, double> data = { { "pi"s, 3.14 }, { "e"s, 2.71 } };  
02  
03 for (const auto& [key, value] : data)  
04     std::cout << key << " - " << value << "\n";
```

# Structured bindings - inicjalizacja wielu zmiennych

```
01 std::vector vec = { 1, 2, 3 };
02
03 for (auto[i, it] = std::tuple{ 0, begin(vec) } ; i < size(vec); ++i, ++it)
04 {
05     std::cout << i << " - " << *it << "\n";
06 }
```

# Tuple-like protocol

## Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:

# Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:
  - `std::tuple_size<E>::value` jest poprawnie skonstruowanym stałym wyrażeniem i ilość identyfikatorów w wiązaniu odpowiada wartości `value`

# Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:
  - std::tuple\_size<E>::value jest poprawnie skonstruowanym stałym wyrażeniem i ilość identyfikatorów w wiązaniu odpowiada wartości value
  - dla każdego identyfikatora wprowadzana jest zmienna, której typem jest "uniwersalna referencja &&" do std::tuple\_element<i, E>::type

# Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:
  - `std::tuple_size<E>::value` jest poprawnie skonstruowanym stałym wyrażeniem i ilość identyfikatorów w wiązaniu odpowiada wartości `value`
  - dla każdego identyfikatora wprowadzana jest zmienna, której typem jest "uniwersalna referencja &&" do `std::tuple_element<i, E>::type`
  - wyrażeniem inicjalizującym każdą wprowadzoną zmienną jest:

# Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:
  - std::tuple\_size<E>::value jest poprawnie skonstruowanym stałym wyrażeniem i ilość identyfikatorów w wiązaniu odpowiada wartości value
  - dla każdego identyfikatora wprowadzana jest zmienna, której typem jest "uniwersalna referencja &&" do std::tuple\_element<i, E>::type
  - wyrażeniem inicjalizującym każdą wprowadzoną zmienną jest:
    - e.get<i>()

# Tuple-like protocol

- Dowolny obiekt e typu E może zostać użyty w wiązaniu structured bindings, jeśli:
  - std::tuple\_size<E>::value jest poprawnie skonstruowanym stałym wyrażeniem i ilość identyfikatorów w wiązaniu odpowiada wartości value
  - dla każdego identyfikatora wprowadzana jest zmienna, której typem jest "uniwersalna referencja &&" do std::tuple\_element<i, E>::type
  - wyrażeniem inicjalizującym każdą wprowadzoną zmienną jest:
    - e.get<i>()
    - lub get<i>(e) (użyte ADL)

# Tuple-like protocol

```
01 enum class Something : uint8_t
02 {
03     Some,
04     Thing
05 };
06
07 const std::unordered_map<uint8_t, std::string_view> something_description
08     {static_cast<uint8_t>(Something::Some), "Something::Some"sv},
09     {static_cast<uint8_t>(Something::Thing), "Something::Thing"sv}};
10
11 const auto [value, description] = Something::Thing;
12
13 assert(value == 1);
14 assert(description == "Something::Thing"s);
```

# Tuple-like protocol (1)

```
01 template <>
02 struct std::tuple_size<Something>
03 {
04     static unsigned int const value = 2;
05 };
```

## Tuple-like protocol (2)

```
01 template <>
02 struct std::tuple_element<0, Something>
03 {
04     using type = uint8_t;
05 };
06
07 template <>
08 struct std::tuple_element<1, Something>
09 {
10     using type = string_view;
11 };
```

# Tuple-like protocol (3)

```
01 template <size_t>
02 auto get(const Something&);
03
04 template <>
05 auto get<0>(const Something& s)
06 {
07     return static_cast<uint8_t>(s);
08 }
09
10 template <>
11 auto get<1>(const Something& s)
12 {
13     return something_description.at(static_cast<uint8_t>(s));
14 }
```

# **if/switch z sekcją inicjującą**

# **if/switch z sekcją inicującą**

Nowa składnia instrukcji wyboru:

```
01 if (init; condition)
02 { }
```

```
01 switch (init; condition)
02 { }
```

# if z sekcją inicjującą

```
01 std::vector data = {1, 42, 3453, 665, 5645 };  
02  
03 if (auto pos = std::find(begin(data), end(data), 665); pos != end(data))  
04 {  
05     std::cout << "Item " << *pos << " has been found\n";  
06 }  
07 else  
08 {  
09     std::cout << "Item has not been found\n";  
10     assert(pos == end(data));  
11 }
```

# if z sekcją inicjującą

```
01 std::vector data = {1, 42, 3453, 665, 5645 };  
02  
03 if (auto pos = std::find(begin(data), end(data), 665); pos != end(data))  
04 {  
05     std::cout << "Item " << *pos << " has been found\n";  
06 }  
07 else  
08 {  
09     std::cout << "Item has not been found\n";  
10     assert(pos == end(data));  
11 }
```

# if z sekcją inicjującą

```
01 std::vector data = {1, 42, 3453, 665, 5645 };  
02  
03 if (auto pos = std::find(begin(data), end(data), 665); pos != end(data))  
04 {  
05     std::cout << "Item " << *pos << " has been found\n";  
06 }  
07 else  
08 {  
09     std::cout << "Item has not been found\n";  
10     assert(pos == end(data));  
11 }
```

# if z sekcją inicjującą

```
01 std::vector data = {1, 42, 3453, 665, 5645 };  
02  
03 if (auto pos = std::find(begin(data), end(data), 665); pos != end(data))  
04 {  
05     std::cout << "Item " << *pos << " has been found\n";  
06 }  
07 else  
08 {  
09     std::cout << "Item has not been found\n";  
10     assert(pos == end(data));  
11 }
```

# switch z sekcją inicjującą

```
01 switch (Gadget&& g = get_gadget(); auto s = g.get_status())
02 {
03     case Status::on:
04         cout << "Gadget is on\n";
05         break;
06     case Status::off:
07         cout << "Gadget is off\n";
08         break;
09     case Status::bad:
10         cout << "Gadget is broken\n";
11         break;
12 }
```

## if z sekcją inicjującą + structured bindings

```
01 std::map<int, string> dict;
02
03 if (auto [pos, is_inserted]
04     = dict.insert(std::pair(42, "fourty two"s)); !is_inserted)
05 {
06     const auto& [key, value] = *pos;
07
08     cout << key << " is already in a dictionary\n";
09 }
```

## **Obiekty tymczasowe w sekcji inicjującej**

Obiekt tymczasowy utworzony na potrzeby inicjalizacji istnieje tylko w obrębie sekcji inicjującej (tak jak w pętli for)!

# Obiekty tymczasowe w sekcji inicjującej (bug)

```
01 if (std::lock_guard<std::mutex>(mtx); !q.empty())
02 {
03     std::cout << q.front() << std::endl;
04 }
```

# Obiekty tymczasowe w sekcji inicjującej (bug)

```
01 if (std::lock_guard<std::mutex>(mtx); !q.empty())
02 {
03     std::cout << q.front() << std::endl;
04 }
```

ERROR! locks ends before ;

# Obiekty tymczasowe w sekcji inicjującej (ok)

```
01 if (std::lock_guard<std::mutex> lk(mtx); !q.empty()) // OK - lock has name  
02 {  
03     std::cout << q.front() << std::endl;  
04 }
```

# Obiekty tymczasowe w sekcji inicjującej (ok)

```
01 if (std::lock_guard<std::mutex> lk(mtx); !q.empty()) // OK - lock has name  
02 {  
03     std::cout << q.front() << std::endl;  
04 }
```

OK! lock\_guard has name

# **constexpr if**

## constexpr if

umożliwia wybór na etapie kompilacji bloku instrukcji `then/else` w zależności od warunku, który jest wyrażeniem `constexpr`:

```
01 if constexpr(condition)
02 {
03     // ...
04 }
05 else
06 {
07     // ...
08 }
```

# constexpr if

umożliwia znaczne uproszczenie kodu szablonów:

```
01 template<class T>
02 auto compute(T x) → enable_if_t<SupportsAPI<T>::value, int>
03 {
04     return optimized_computation(x);
05 }
06
07 template<class T>
08 auto compute(T x) → enable_if_t<!SupportsAPI<T>::value, int>
09 {
10     return generic_computation(x);
11 }
```

# constexpr if

```
01 template<class T>
02 auto compute(T x)
03 {
04     if constexpr(SupportsAPI<T>::value)
05     {
06         return optimized_computation(x);
07     }
08     else
09     {
10         return generic_computation(x);
11     }
12 }
```

# Statyczne składowe inline

# Statyczne składowe inline

- Statyczne składowe klasy oznaczone jako `inline` są uznawane jako definicja takiej zmiennej w programie
- Gwarantowana jest jednokrotna definicja zmiennej nawet wtedy, gdy nagłówek z definicją jest włączany w wielu jednostkach translacji
- Nie ma potrzeby tworzenia pliku `.cpp` tylko na potrzeby definicji pól statycznych

# Statyczne składowe inline

Plik: gadget.hpp

```
01 #include <string>
02
03 class Gadget
04 {
05 public:
06     static inline const std::string class_id{"Gadget"};
07
08     static size_t count()
09     {
10         return counter_;
11     }
12 private:
13     static inline size_t counter_ = 0;
14 };
```

# Statyczne składowe inline

Plik: main.cpp

```
01 #include "gadget.hpp"
02 #include <iostream>
03
04 int main()
05 {
06     std::cout << "No of gadgets: " << Gadget::count() << std::endl;
07 }
```

# Agregaty w C++17

# Agregaty w C++17

C++17 rozszerza definicję agregatu:

- agregaty w C++17 mogą posiadać klasy bazowe, po których dziedziczą publicznie
- inicjalizacja jest możliwa za pomocą zagnieżdżonych klamer {}
- biblioteka standardowa dostarcza nową cechę (trait) - `std::is_aggregate<T>`

# Definicja agregatu w C++17

C++17 definiuje agregat jako:

- tablicę
- lub klasę(class, struct, lub union), która:
  - nie posiada konstruktorów explicit lub zdefiniowanych przez użytkownika
  - nie posiada konstruktorów odziedziczonych deklaracją using
  - nie posiada prywatnych lub chronionych niestatycznych danych składowych
  - nie posiada wirtualnych funkcji składowych
  - nie posiada wirtualnych, prywatnych lub chronionych klas bazowych
- inicjalizacja agregatu, nie może wykorzystywać prywatnych lub chronionych konstruktorów klasy bazowej

# Agregaty

```
01 struct Base1
02 {
03     int b1;
04     int b2 = 42;
05 };
06
07 struct Base2
08 {
09     Base2()
10     {
11         b3 = 42;
12     }
13
14     int b3;
15 };
```

# Agregaty

```
01 struct Derived : Base1, Base2
02 {
03     int d;
04 }
05
06 Derived d1{ {1, 2}, {}, 4 }; // d1.b1 = 1, d1.b2 = 2, d1.b3 = 42, d1.d =
07 Derived d2{ {}, {}, 4 }; // d2.b1 = 0, d2.b2 = 42, d2.b3 = 42, d2.d = 4
```

# Aggregaty

```
01 template <typename T>
02 struct Aggregate : std::string, std::complex<T>
03 {
04     std::string data;
05 };
06
07 Aggregate<double> agg1{ {"aggregate"}, {4.5, 6.7}, "test" };
```

# Atrybuty

# Atrybut **[[nodiscard]]**

- wymusza zgłoszenie ostrzeżenia w przypadku, gdy zwracana wartość nie jest użyta

```
01 template <typename F, typename... Args>
02 [[nodiscard]] future<decltype(F())> async(F&& f, Args&&...);
```

## Atrybut [[maybe\_unused]]

- dezaktywuje ostrzeżenia o nieużywanej zmiennej, jeśli taka jest intencja programisty

```
01 [[maybe_unused]] int x = foo();
```

# Atrybut [[fallthrough]]

- używany w instrukcji switch, gdy wybrana etykieta case zawiera instrukcje, ale nie kończy się instrukcją break
- musi poprzedzać inną etykietę case (jeśli nie, kod jest illformed)

```
01 void f(int n)
02 {
03     switch (n) {
04         case 1:
05         case 2:
06             step1();
07             [[fallthrough]];
08         case 3: // no warning on fallthrough
09             step2();
10         case 4: // compiler may warn on fallthrough
11             step3();
12             [[fallthrough]]; // illformed, not before a case label
13     }
14 }
```

# Atrybut [[deprecated]]

- może być stosowany dla przestrzeni nazw oraz wyliczeń:

```
01 enum Coffee {  
02     espresso = 1,  
03     americano [[deprecated]] = espresso  
04 };  
05  
06 namespace [[deprecated]] LegacyCode  
07 {  
08     // ...  
09 }
```

# Zagnieżdżone przestrzenie nazw

# Zagnieżdżone przestrzenie nazw

- Dozwolona jest nowa składnia przy zagnieżdżaniu przestrzeni nazw:
- Zamiast:

```
01 namespace A {  
02     namespace B {  
03         namespace C {  
04             // ...  
05         }  
06     }  
07 }
```

- można napisać:

```
01 namespace A::B::C {  
02     // ...  
03 }
```

# Statyczne asercje bez komunikatów o błędach

- Od C++17 `static_assert()` nie wymaga przekazania komunikatu o błędzie. Jeśli asercja nie jest zaliczona, wyświetlany jest komunikat domyślny:

```
01 static_assert(sizeof(int) ≥ 4, "ints are too small"); // OK since C++11
02 static_assert(sizeof(int) ≥ 4); // OK since C++17
```

# Lamby w C++17

# Przechwytywanie this

- Można użyć trzech opcji, aby przechwycić wskaźnik `this` w funkcjach składowych:

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10     }
11 };
```

# Przechwytywanie this

- Można użyć trzech opcji, aby przechwycić wskaźnik this w funkcjach składowych:

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10     }
11 };
```

OK - this captured implicitly by =

# Przechwytywanie this

- Można użyć trzech opcji, aby przechwycić wskaźnik `this` w funkcjach składowych:

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10     }
11 };
```

OK - this captured implicitly by &

# Przechwytywanie this

- Można użyć trzech opcji, aby przechwycić wskaźnik `this` w funkcjach składowych:

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10     }
11 };
```

OK - this captured explicitly

# Przechwytywanie kopii obiektu

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10         execute([*this] { std::cout << name_ << std::endl; }) // OK since ()
11     }
12 };
```

# Przechwytywanie kopii obiektu

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap-
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap-
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10         execute([*this] { std::cout << name_ << std::endl; }) // OK since C
11     }
12 };
```

OK - this captured implicitly by =

# Przechwytywanie kopii obiektu

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10         execute([*this] { std::cout << name_ << std::endl; }) // OK since C
11     }
12 };
```

OK - this captured implicitly by &

# Przechwytywanie kopii obiektu

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute[this] { std::cout << name_ << std::endl; }); // OK - this
10         execute[*this] { std::cout << name_ << std::endl; }) // OK since C
11     }
12 };
```

OK - this captured explicitly

# Przechwytywanie kopii obiektu

```
01 class Gadget
02 {
03     std::string name_;
04 public:
05     void do_sth()
06     {
07         execute([&] { std::cout << name_ << std::endl; }); // OK - this cap
08         execute([=] { std::cout << name_ << std::endl; }); // OK - this cap
09         execute([this] { std::cout << name_ << std::endl; }); // OK - this
10         execute([*this] { std::cout << name_ << std::endl; }) // OK since ()
11     }
12 };
```

OK - captured copy of object

# Lamby `constexpr`

- Od C++17 wyrażenia lambda są traktowane domyślnie jako wyrażenia `constexpr` (jeśli jest to możliwe):

```
01 auto squared = [](auto x) { return x * x; } // implicitly constexpr
02
03 std::array<int, squared(8)> buffer{};
04
05 static_assert(buffer.size() == 64);
```

# Lambdy `constexpr`

- Można explicite zastosować również słowo kluczowe `constexpr` w definicji lambdy:

```
01 auto squared = [](auto x) constexpr {  
02     return x * x;  
03 };
```

# Zdefiniowana kolejność ewaluacji wyrażeń

# Niezdefiniowana kolejność ewaluacji wyrażeń

- Kolejność ewaluacji wyrażeń dla wielu operatorów w C++ była niezdefiniowana w standardzie.
- W efekcie przewidzenie wyniku wykonania operacji korzystającej z takich operatorów było w zasadzie niemożliwe.

# Niezdefiniowana kolejność ewaluacji wyrażeń

# Niezdefiniowana kolejność ewaluacji wyrażeń

```
01 std::map<int, int> dict;  
02 dict[0] = dict.size(); // after statement: dict = { {0, 0} } or { { 0, 1} }
```

# Niezdefiniowana kolejność ewaluacji wyrażeń

```
01 std::map<int, int> dict;
02 dict[0] = dict.size(); // after statement: dict = { {0, 0} } or { {0, 1} }
```

```
01 std::string s = "but I have heard it works even if you don't believe in it";
02
03 s.replace(0, 4, "")
04 .replace(s.find("even"), 4, "only")
05 .replace(s.find(" don't"), 6, "");
06
07 assert(s == "I have heard it works only if you believe in it"); // it may fail
```

# Niezdefiniowana kolejność ewaluacji wyrażeń

```
01 std::map<int, int> dict;
02 dict[0] = dict.size(); // after statement: dict = { {0, 0} } or { {0, 1} }
```

```
01 std::string s = "but I have heard it works even if you don't believe in it";
02
03 s.replace(0, 4, "")
04 .replace(s.find("even"), 4, "only")
05 .replace(s.find(" don't"), 6, "");
06
07 assert(s == "I have heard it works only if you believe in it"); // it may fail
```

```
01 std::cout << f() << g() << h(); // UB: undefined evaluation of order
02 std::cout.operator<<(f()).operator<<(g()).operator<<(h());
```

# Reguły ewaluacji wyrażeń w C++17

- Wyrażenia postfix są ewaluowane od lewej do prawej.
  - Dotyczy to również wywołań funkcji i wyrażeń związań z wyborem składowej klasy (struktury).
- Wyrażenia przypisania są ewaluowane od prawej do lewej.
- Operandy dla operatorów przesunięć są ewaluowane od lewej do prawej
- Kolejność ewaluacji wyrażeń zawierających przeciążone operatory jest określona tak jak w przypadku odpowiednich operatorów wbudowanych, a nie przez reguły związane z wywołaniami funkcji

# Reguły ewaluacji wyrażeń w C++17

- W rezultacie następujące wyrażenia są ewaluowane w kolejności a, potem b:

```
01  a.b
02  a→b
03  a→*b
04  a(b1, b2, b3)
05  b @= a
06  a[b]
07  a << b
08  a >> b
```

# Reguły ewaluacji wyrażeń w C++17

- W rezultacie wyrażenie:

```
01 s.replace(0, 4, "") // 1-st  
02 .replace(s.find("even"), 4, "only") // 2nd  
03 .replace(s.find(" don't"), 6, ""); // 3rd
```

jest ewaluowane w określony sposób, ale kolejność ewaluacji argumentów wywołań metody `replace` wciąż nie jest specyfikowana przez standard

# Iinicjalizacja typów wyliczeniowych

# Inicjalizacja typów wyliczeniowych

- Dla typów wyliczeniowych z określonym typem całkowitym (*fixed underlying type*) standard C++17 dopuszcza bezpośrednią inicjalizację listową (*direct list initialization*) wartością całkowitą
- Dotyczy to zarówno klasycznych wyliczeń *enum*, jak i wprowadzonych w C++11 wyliczeń *enum class*
- Dla klasycznych typów wyliczeniowych *enum*, bez określonego typu całkowitego, taka inicjalizacja wciąż jest traktowana jako błąd komilacji

# enum class

```
01 enum class Coffee { espresso, cappuccino, chemex };
02
03 // Coffee c1 = 0; // ERROR (all versions)
04 // Coffee c2(0); // ERROR (all versions)
05 Coffee c3{0}; // OK since C++17
06
07
08 enum class GuitarType : char { stratocaster, les_paul };
09
10 // GuitarType gt1 = 1; // ERROR (all versions)
11 // GuitarType gt2(1); // ERROR (all versions)
12 GuitarType gt3{1}; // OK since C++17
```

# enum

```
01 enum EngineType { diesel, petrol, wankel };
02
03 // EngineType e1 = 0; // ERROR (all versions)
04 // EngineType e2(0); // ERROR (all versions)
05 // EngineType e3{2}; // ERROR (all versions)
06
07
08 enum MovieFormat : char { divx, mpeg };
09
10 // MovieFormat mv1 = 1; // ERROR (all versions)
11 // MovieFormat mv2(1); // ERROR (all versions)
12 MovieFormat mv3{1}; // OK since C++17
```

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3);  
11  
12 std::cout << std::to_integer<int>(b);
```

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3);  
11  
12 std::cout << std::to_integer<int>(b);
```

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3);  
11  
12 std::cout << std::to_integer<int>(b);
```

Inicjalizacja zmiennej

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3);  
11  
12 std::cout << std::to_integer<int>(b);
```

Operatory bitowe '&', '|', '^', '~', '<<', '>>' etc. są zdefiniowane dla std::byte

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3) ;  
11  
12 std::cout << std::to_integer<int>(b);
```

Podobnie jak operatory '&=' , '|=' , '^=' , '<<=' , '>>='

# Definiowanie nowych typów całkowitych za pomocą wyliczeń

- Od C++17 można użyć typu wyliczeniowego do zdefiniowania nowego typu całkowitego niepodlegającego niejawnym konwersjom:

```
01 enum class std::byte : unsigned char {};  
02  
03 std::byte b{42};  
04  
05 // std::byte ill_formed{-22}; // ERROR  
06 // auto result = b + 24; // ERROR  
07  
08 std::byte bits = b & std::byte{13};  
09  
10 bits |= (b << 3);  
11  
12 std::cout << std::to_integer<int>(b);
```

Konwersja do typu całkowitego za pomocą funkcji `std::to_integer<Integral>(b)`

# Iinicjalizacja {} i auto

# Inicjalizacja {} i auto

- Zmieniona została reguła dotycząca automatycznej detekcji typu w przypadku inicjalizacji bezpośredniej za pomocą inicjalizacji listowej

# Inicjalizacja {} i auto (C++11/14)

```
01 int x1(42); // direct initialization with C++98/03 syntax
02 int x2{42}; // direct initialization with C++11
03 int x3 = 665; // copy initialization
```

```
01 auto a1(42); // direct initialization → int
02 auto a2{42}; // direct initialization → initializer_list<int>
03 auto a3{42, 665}; // direct initialization → initializer_list<int>
04
05 auto a4 = 42; // copy initialization → int
06 auto a5 = {42}; // copy initialization → initializer_list<int>
07 auto a6 = {42, 665}; // copy initialization → initializer_list<int>
```

# Inicjalizacja {} i auto (C++17)

```
01 int x1(42); // direct initialization with C++98/03 syntax
02 int x2{42}; // direct initialization with C++11
03 int x3 = 665; // copy initialization
```

```
01 auto a1(42); // direct initialization → int
02 auto a2{42}; // direct initialization → int (new rule!!!)
03 auto a3{42, 665}; // ERROR
04
05 auto a4 = 42; // copy initialization → int
06 auto a5 = {42}; // copy initialization → initializer_list<int>
07 auto a6 = {42, 665}; // copy initialization → initializer_list<int>
```

# Inicjalizacja {} i auto (C++17)

```
01 int x1(42); // direct initialization with C++98/03 syntax
02 int x2{42}; // direct initialization with C++11
03 int x3 = 665; // copy initialization
```

```
01 auto a1(42); // direct initialization → int
02 auto a2{42}; // direct initialization → int (new rule!!!)
03 auto a3{42, 665}; // ERROR
04
05 auto a4 = 42; // copy initialization → int
06 auto a5 = {42}; // copy initialization → initializer_list<int>
07 auto a6 = {42, 665}; // copy initialization → initializer_list<int>
```

# Inicjalizacja {} i auto (C++17)

```
01 int x1(42); // direct initialization with C++98/03 syntax
02 int x2{42}; // direct initialization with C++11
03 int x3 = 665; // copy initialization
```

```
01 auto a1(42); // direct initialization → int
02 auto a2{42}; // direct initialization → int (new rule!!!)
03 auto a3{42, 665}; // ERROR
04
05 auto a4 = 42; // copy initialization → int
06 auto a5 = {42}; // copy initialization → initializer_list<int>
07 auto a6 = {42, 665}; // copy initialization → initializer_list<int>
```

# **noexcept jako część typu funkcji**

# noexcept jako część typu funkcji

- System typów w C++17 uwzględnia specyfikację noexcept dla funkcji:

```
01 void func1();
02 void func2() noexcept;
03
04 static_assert(is_same_v<decltype(func1), decltype(func2)>); // ERROR - dif
05
06 void (*fp)() noexcept;
07
08 fp = func2(); // OK
09 fp = func1(); // ERROR since C++17
```

# noexcept jako część typu funkcji

- Zmiana ta może spowodować, że kod z C++14 może się nie skompilować w C++17:

```
01 template <typename F>
02 void call(F f1, F f2)
03 {
04     f1();
05     f2();
06 }
07
08 call(func1, func2); // ERROR since C++17
```

# **Class template argument deduction (CTAD)**

# Class template argument deduction (CTAD)

- Mechanizm dedukcji argumentów szablonu klasy
- Typy parametrów szablonu klasy mogą być dedukowane na podstawie argumentów przekazanych do konstruktora tworzonego obiektu

# CTAD

```
01 template <typename T1, typename T2>
02 struct ValuePair
03 {
04     T1 fst;
05     T2 snd;
06
07
08     ValuePair(T1 f, T2 s) : fst{f}, snd{s}
09     {
10     }
11 };
```

# CTAD

```
01 template <typename T1, typename T2>
02 struct ValuePair
03 {
04     T1 fst;
05     T2 snd;
06
07     template <typename T1, typename T2>
08     ValuePair(T1 f, T2 s) : fst{f}, snd{s}
09     {
10     }
11 };
```

# CTAD

```
01 template <typename T1, typename T2>
02 struct ValuePair
03 {
04     T1 fst;
05     T2 snd;
06
07     template <typename T1, typename T2>
08     ValuePair(T1 f, T2 s) : fst{f}, snd{s}
09     {
10     }
11 };
```

Takie podejście umożliwia dedukcję parametrów szablonu

# CTAD

```
01 template <typename T1, typename T2>
02 struct ValuePair {
03     T1 fst; T2 snd;
04
05     ValuePair(T1 f, T2 s);
06 };
07
08 ValuePair<int, double> vp1(1, 3.14); // OK - all versions of standard
09
10 ValuePair vp2(1, 3.14); // deduces ValuePair<int, double>
11 ValuePair vp3{1, 3.14}; // deduced ValuePair<int, double>
12
13 auto vp4 = ValuePair(1, "text"); // deduces ValuePair<int, const char*>
14 auto vp5 = ValuePair{3.14, "pi"s}); // deduces ValuePair<int, std::string>
```

# CTAD

- Nie można częściowo dedukować argumentów szablonu klasy.
- Należy wyspecyfikować lub wydedukować wszystkie parametry z wyjątkiem parametrów domyślnych

# CTAD

```
01 template <typename T1 = int, typename T2 = T1>
02 struct Generic {
03     explicit Generic(T1 f = T1{}, T2 s = T2{});
04 };
05
06 Generic<int, double> g0{1, 3.14}; // no deduction
07
08 Generic g1{1, 3.14};
09 static_assert(is_same_v<decltype(g1), Generic<int, double>>);
10
11 Generic<double> g2{3.14, 1}; // no deduction
12 static_assert(is_same_v<decltype(g2), Generic<double, double>>);
13
14 Generic g3{3.14};
15 static_assert(is_same_v<decltype(g3), Generic<double, double>>);
16
17 Generic<> g4; // no deduction
18 static_assert(is_same_v<decltype(g4), Generic<int, int>>);
19
```

# CTAD - specjalny przypadek dedukcji

- Jeżeli kod służący do dedukcji argumentów szablonu klasy może być zinterpretowany jako przypadek inicjalizacji poprzez kopię, to kompilator preferuje taką interpretację:

```
01 std::vector v{1, 2, 3}; // vector<int>
02
03 std::vector data1{v, v}; // vector<vector<int>>
04
05 std::vector data2{v}; // vector<int>!
```

# CTAD - specjalny przypadek dedukcji

- Jeżeli kod służący do dedukcji argumentów szablonu klasy może być zinterpretowany jako przypadek inicjalizacji poprzez kopię, to kompilator preferuje taką interpretację:

```
01 std::vector v{1, 2, 3}; // vector<int>
02
03 std::vector data1{v, v}; // vector<vector<int>>
04
05 std::vector data2{v}; // vector<int>!
```

W powyższym kodzie dedukcja argumentów szablonu `vector` zależy od ilości argumentów przekazanych do konstruktora!

# Podpowiedzi dedukcyjne (deduction guides)

- C++17 umożliwia tworzenie podpowiedzi dla kompilatora, jak powinny być dedukowane typy parametrów szablonu klasy na podstawie wywołania odpowiedniego konstruktora
- Daje to możliwość poprawy/modyfikacji domyślnego procesu dedukcji

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

```
01 Data x{12}; // OK → Data<int> x{12};
02 Data y(12.0); // OK → Data<int> y(12);
03 auto z = Data{12}; // OK → auto z = Data<int>{12};
```

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

```
01 Data x{12}; // OK → Data<int> x{12};
02 Data y(12.0); // OK → Data<int> y(12);
03 auto z = Data{12}; // OK → auto z = Data<int>{12};
```

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

```
01 Data x{12}; // OK → Data<int> x{12};
02 Data y(12.0); // OK → Data<int> y(12);
03 auto z = Data{12}; // OK → auto z = Data<int>{12};
```

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

```
01 Data x{12}; // OK → Data<int> x{12};
02 Data y(12.0); // OK → Data<int> y(12);
03 auto z = Data{12}; // OK → auto z = Data<int>{12};
```

# Podpowiedzi dedukcyjne (deduction guides)

- Podpowiedź dedukcyjna musi zostać umieszczona w tym samym zakresie (przestrzeni nazw) i może mieć postać:

```
01 template <typename T>
02 class Data
03 {
04     T value;
05 public:
06     Data(const T& v) : value(v)
07     {}
08 };
09
10 template <typename T>
11 Data(T) → Data<T>;
```

```
01 Data x{12}; // OK → Data<int> x{12};
02 Data y(12.0); // OK → Data<int> y(12);
03 auto z = Data{12}; // OK → auto z = Data<int>{12};
```

# Deduction guides

```
01 [explicit] template-name(parameter-decl-clause) → simple-template-id;
```

```
01 template <typename T>
02 struct S
03 {
04     T data;
05 };
06
07 template<class U>
08 S(U) → S<typename U::type>;
09
10 struct A {
11     using type = short;
12     operator type();
13 };
14
15 S x{A()}; // x is of type S<short, int>
```

# Deduction guides

```
01 [explicit] template-name(parameter-decl-clause) → simple-template-id;
```

```
01 template <typename T>
02 struct S
03 {
04     T data;
05 };
06
07 template<class U>
08 S(U) → S<typename U::type>;
09
10 struct A {
11     using type = short;
12     operator type();
13 };
14
15 S x{A()}; // x is of type S<short, int>
```

# Deduction guides

- Podpowiedzi dedukcyjne nie są szablonami funkcji
  - służą jedynie dedukowaniu argumentów szablonu i nie są wywoływane
  - przekazanie w podpowiedzi argumentu przez wartość nie ma znaczenia dla wydajności

```
01 template <typename T>
02 struct X
03 {
04     //...
05 };
06
07 template <typename T>
08 struct Y
09 {
10     Y(const X<T>&);
11     Y(X<T>&&);
12 };
13
14 template <typename T> Y(X<T>) → Y<T>;
```

# Deduction guides

- Podpowiedzi dedukcyjne nie są szablonami funkcji
  - służą jedynie dedukowaniu argumentów szablonu i nie są wywoływane
  - przekazanie w podpowiedzi argumentu przez wartość nie ma znaczenia dla wydajności

```
01 template <typename T>
02 struct X
03 {
04     //...
05 };
06
07 template <typename T>
08 struct Y
09 {
10     Y(const X<T>&);
11     Y(X<T>&&);
12 };
13
14 template <typename T> Y(X<T>) → Y<T>;
```

# Podpowiedzi dedukcyjne vs. Konstruktory

- Podpowiedzi dedukcyjne rywalizują z konstruktorami klasy
- Mechanizm dedukcji wykorzystuje konstruktor lub podpowiedź, która ma najwyższy priorytet zgodnie z regułami przeciążania funkcji
- Jeśli konstruktor i podpowiedź pasują jednakowo dobrze, kompilator preferuje podpowiedź dedukcyjną

# Podpowiedzi dedukcyjne vs. Konstruktory

```
01 template <typename T>
02 struct Thing
03 {
04     Thing(const T&)
05     {
06     }
07 };
08
09 Thing(int) → Thing<long>;
```

```
01 Thing t1{42}; // T deduced as long
02 Thing t2{'a'}; // T deduced as char
```

# Podpowiedzi dedukcyjne vs. Konstruktory

```
01 template <typename T>
02 struct Thing
03 {
04     Thing(const T&)
05     {
06     }
07 };
08
09 Thing(int) → Thing<long>;
```

```
01 Thing t1{42}; // T deduced as long
02 Thing t2{'a'}; // T deduced as char
```

# Podpowiedzi dedukcyjne vs. Konstruktory

```
01 template <typename T>
02 struct Thing
03 {
04     Thing(const T&)
05     {
06     }
07 };
08
09 Thing(int) → Thing<long>;
```

```
01 Thing t1{42}; // T deduced as long
02 Thing t2{'a'}; // T deduced as char
```

# Niejawne podpowiedzi dedukcyjne

- Ponieważ często podpowiedź dedukcyjna jest potrzebna dla każdego konstruktora klasy, standard C++17 wprowadza mechanizm niejawnych podpowiedzi dedukcyjnych (*implicit deduction guides*)

# Niejawne podpowiedzi dedukcyjne

- Lista parametrów szablonu dla podpowiedzi zawiera listę parametrów z szablonu klasy - w przypadku szablonowego konstruktora klasy kolejnym elementem jest lista parametrów szablonu konstruktora klasy
- Parametry „funkcyjne” podpowiedzi są kopiowane z konstruktora lub konstruktora szablonowego
- Zalecany typ w podpowiedzi jest nazwą szablonu z argumentami, które są parametrami szablonu wziętymi z klasy szablonowej

# Implicit deduction guides

```
01 template <typename T>
02 class S
03 {
04     private:
05         T value;
06     public:
07         S(T v);
08
09     template <typename U>
10     S(T v, U u);
11 };
```

niejawne podpowiedzi dedukcyjne będą wyglądać następująco:

```
01 template <typename T> S(T) → S<T>;
02
03 template <typename T, typename U> S(T, U) → S<T>;
```

# CTAD - agregaty

Jeśli szablon klasy jest agregatem, to mechanizm automatycznej dedukcji argumentów szablonu wymaga napisania jawnej podpowiedzi dedukcyjnej:

```
01 template <typename T>
02 struct Aggregate1
03 {
04     T value;
05 };
06
07 Aggregate1 agg1{8}; // ERROR
08 Aggregate1 agg2{"eight"}; // ERROR
09 Aggregate1 agg3 = 3.14; // ERROR
```

# CTAD - agregaty

Gdy napiszemy dla agregatu podpowiedź, to możemy zacząć korzystać z mechanizmu dedukcji:

```
01 template <typename T>
02 struct Aggregate2
03 {
04     T value;
05 };
06
07 template <typename T>
08 Aggregate2(T) → Aggregate2<T>;
09
10 Aggregate2 agg1{8}; // OK → Aggregate2<int>
11 Aggregate2 agg2{"eight"}; // OK → Aggregate2<const char*>
12 Aggregate2 agg3 = { 3.14 }; // OK → Aggregate2<double>
```

# Podpowiedzi dedukcyjne w bibliotece standardowej

# std::pair

```
01 template<class T1, class T2>
02 pair(T1, T2) → pair<T1, T2>;
03
04 pair p1(1, 3.14); // → pair<int, double>
05
06 pair p2{3.14f, "text"s}; // → pair<float, string>
07
08 pair p3{3.14f, "text"}; // → pair<float, const char*>
09
10 int tab[3] = { 1, 2, 3 };
11 pair p4{1, tab}; // → pair<int, int*>
```

# std::pair

```
01 template<class T1, class T2>
02 pair(T1, T2) → pair<T1, T2>;
03
04 pair p1(1, 3.14); // → pair<int, double>
05
06 pair p2{3.14f, "text"s}; // → pair<float, string>
07
08 pair p3{3.14f, "text"}; // → pair<float, const char*>
09
10 int tab[3] = { 1, 2, 3 };
11 pair p4{1, tab}; // → pair<int, int*>
```

# std::pair

```
01 template<class T1, class T2>
02 pair(T1, T2) → pair<T1, T2>;
03
04 pair p1(1, 3.14); // → pair<int, double>
05
06 pair p2{3.14f, "text"s}; // → pair<float, string>
07
08 pair p3{3.14f, "text"}; // → pair<float, const char*>
09
10 int tab[3] = { 1, 2, 3 };
11 pair p4{1, tab}; // → pair<int, int*>
```

# std::pair

```
01 template<class T1, class T2>
02 pair(T1, T2) → pair<T1, T2>;
03
04 pair p1(1, 3.14); // → pair<int, double>
05
06 pair p2{3.14f, "text"s}; // → pair<float, string>
07
08 pair p3{3.14f, "text"}; // → pair<float, const char*>
09
10 int tab[3] = { 1, 2, 3 };
11 pair p4{1, tab}; // → pair<int, int*>
```

# std::pair

```
01 template<class T1, class T2>
02 pair(T1, T2) → pair<T1, T2>;
03
04 pair p1(1, 3.14); // → pair<int, double>
05
06 pair p2{3.14f, "text"s}; // → pair<float, string>
07
08 pair p3{3.14f, "text"}; // → pair<float, const char*>
09
10 int tab[3] = { 1, 2, 3 };
11 pair p4{1, tab}; // → pair<int, int*>
```

# std::tuple

```
01 template<class... UTypes>
02 tuple(UTypes...) → tuple<UTypes...>;
03
04 template<class T1, class T2>
05 tuple(pair<T1, T2>) → tuple<T1, T2>;
06
07 int x = 10;
08 const int& cref_x = x;
09 tuple t1{x, &x, cref_x, "hello", "world"s}; // → tuple<int, int*, int, const char*, string>
10
11 pair p{1, "one"s};
12 tuple t2{p}; // → tuple<int, string>
```

# std::tuple

```
01 template<class... UTypes>
02 tuple(UTypes...) → tuple<UTypes...>;
03
04 template<class T1, class T2>
05 tuple(pair<T1, T2>) → tuple<T1, T2>;
06
07 int x = 10;
08 const int& cref_x = x;
09 tuple t1{x, &x, cref_x, "hello", "world"s}; // → tuple<int, int*, int, const char*, string>
10
11 pair p{1, "one"s};
12 tuple t2{p}; // → tuple<int, string>
```

# std::tuple

```
01 template<class... UTypes>
02 tuple(UTypes...) → tuple<UTypes...>;
03
04 template<class T1, class T2>
05 tuple(pair<T1, T2>) → tuple<T1, T2>;
06
07 int x = 10;
08 const int& cref_x = x;
09 tuple t1{x, &x, cref_x, "hello", "world"s}; // → tuple<int, int*, int, const char*, string>
10
11 pair p{1, "one"s};
12 tuple t2{p}; // → tuple<int, string>
```

# std::optional

```
01 template<class T> optional(T) → optional<T>;  
02  
03 optional o1(3); // → optional<int>  
04 optional o2 = o1; // → optional<int>
```

# Inteligentne wskaźniki

- Dedukcja dla argumentów konstruktora będących wskaźnikami jest zablokowana:

```
01 int* ptr = new int{5};  
02 std::unique_ptr uptr{ip}; // ERROR - ill-formed (due to array type clash)  
03 std::shared_ptr sptr{ip}; // ERROR - ill-formed (due to array type clash)
```

# Inteligentne wskaźniki

Wspierana jest dedukcja przy konwersjach

- z `weak_ptr/unique_ptr` do `shared_ptr`:

```
01 template <class T> shared_ptr(weak_ptr<T>) → shared_ptr<T>;  
02 template <class T, class D> shared_ptr(unique_ptr<T, D>) → shared_ptr<T>
```

- z `shared_ptr` do `weak_ptr`

```
01 template<class T> weak_ptr(shared_ptr<T>) → weak_ptr<T>;
```

# Inteligentne wskaźniki (CTAD)

```
01 auto uptr = std::make_unique<int>(3);
02
03 std::shared_ptr sptr = std::move(uptr); // → shared_ptr<int>
04
05 std::weak_ptr wptr = sptr; // → weak_ptr<int>
06
07 std::shared_ptr sptr2{wptr}; // → shared_ptr<int>
```

# std::function

```
01 int add(int x, int y)
02 {
03     return x + y;
04 }
05
06 std::function f1 = &add; // std::function<int(int, int)>
07 assert(f1(4, 5) == 9);
08
09 std::function f2 =
10     [] (const string& txt) { std::cout << txt << " from lambda!\n"; };
11 f2("Hello");
```

# Kontenery i CTAD

Dla kontenerów standardowych dozwolona jest dedukcja typu elementu:

- na podstawie typu listy inicjalizacyjnej:

```
01 std::vector vec{1, 2, 3}; // → std::vector<int>
02 std::list lst = {"one"s, "two"s, "three"s}; // → std::list<std::string>
```

- na podstawie typu pary iteratorów przekazanych do konstruktora:

```
01 std::vector vec = { 1, 2, 3 };
02 std::list lst(vec.begin(), vec.end()); // → std::list<int>
```

# Parametry szablonu nie będące typami ze specyfikatorem auto

# Parametry szablonu nie będące typami ze specyfikatorem auto

- C++17 wprowadza możliwość zadeklarowania parametru szablonu nie będącego typem jako `auto` lub `decltype(auto)`.
- W rezultacie typ stałej jest automatycznie dedukowany wg odpowiedniego mechanizmu.

# Parametry szablonu nie będące typami ze specyfikatorem auto

```
01 template <auto N>
02 struct Value
03 {
04     constexpr static auto value{N};
05 };
06
07 Value<42> v1; // → N in S is int
08 Value<'a'> v2; // → N in S is char
09 Value<3.14> v3; // ERROR - template parametr type still cannot be double
10
11 // partial specialization
12 template <uint16_t N> struct Value<N>
13 {
14 };
```

# Parametry szablonu nie będące typami ze specyfikatorem auto

```
01 // list of heterogenous constant template arguments
02 template <auto... Ns> struct ValueList { };
03
04 using HeterogenousContainer = ValueList<665, 42u, nullptr, 'a'>;
```

```
01 // list of homogenous constant template arguments
02 template <auto N, decltype(N)... Ns> struct ValueList { };
03
04 using HomogenousContainer = ValueList<665, 42, 0, 65>;
```

# Parametry szablonu nie będące typami ze specyfikatorem decltype(auto)

```
01 template <decltype(auto) N>
02 struct S {
03     void print() {
04         std::cout << "N has value: " << N << "\n";
05     }
06 };
07
08 constexpr auto x = 665;
09 int y{};
10
11 S<x> s0; // N is int
12 S<(y)> s1; // N is int&
13
14 y = 77;
15
16 S<(y)> s2; // N is int& ⇒ prints: 'N has value 77'
17
18 y = 88;
19
20 s1.print(); // prints: 'N has value 88'
21 s2.print(); // prints: 'N has value 88'
```

# Wyrażenia fold

# Fold expressions

- Koncept redukcji jest jednym z podstawowych pojęć w językach funkcyjonalnych.
- **Fold** w językach funkcyjonalnych to rodzina funkcji wyższego rzędu zwana również **reduce**, **accumulate**, **compress** lub **inject**
- Funkcje fold przetwarzają rekurencyjnie uporządkowane kolekcje danych (listy) w celu zbudowania końcowego wyniku przy pomocy funkcji (operatora) łączącej elementy

# Fold expressions

Dwie najbardziej popularne funkcje z tej rodziny to:

- **fold (fold left)**

```
01 fold (+) 0 [1..5]
02
03 (((((0 + 1) + 2) + 3) + 4) + 5)
```

- **foldr (fold right)**

```
01 foldr (+) 0 [1..5]
02
03 (1 + (2 + (3 + (4 + (5 + 0)))))
```

# Redukcja w C++98 - std::accumulate

```
01 #include <vector>
02 #include <numeric>
03 #include <string>
04
05 using namespace std::string_literals;
06
07 std::vector<int> vec = {1, 2, 3, 4, 5};
08
09 std::accumulate(std::begin(vec), std::end(vec), "0"s,
10                 [] (const std::string& reduced, int item) {
11                     return "("s + reduced + " " + "s + std::to_string(item) + ")"s;
12                 });
13
```

```
01 (((((0 + 1) + 2) + 3) + 4) + 5)
```

# Variadic templates & head-tail idiom

- Implementacja redukcji z wykorzystaniem *variadic templates* często wykorzystuje idiom *Head-Tail*, który polega na rekursywnym wywołaniu funkcji z częścią parametrów
- Rekurencja jest przerywana przez implementację funkcji albo przez specjalizację szablonu klasy dla końcowego przypadku

# Variadic templates & Head-Tail idiom

```
01 template <typename T>
02 auto sum(const T& item)
03 {
04     return item;
05 }
06
07 template <typename Head, typename... Tail>
08 auto sum(const Head& head, const Tail&... tail)
09 {
10     return head + sum(tail...);
11 }
12
13 auto result = sum(1, 2, 3, 4, 5, 6);
```

# Fold expressions w C++17

- Wyrażenia typu fold umożliwiają uproszczenie rekurencyjnych implementacji dla zmiennej liczby argumentów szablonu.

```
01 template <typename... TArgs>
02 auto sum(const TArgs&... args)
03 {
04     return (... + args);
05 }
06
07 auto result = sum(1, 2, 3, 4, 5);
```

# Fold expressions w C++17

- Wyrażenia typu fold umożliwiają uproszczenie rekurencyjnych implementacji dla zmiennej liczby argumentów szablonu.

```
01 template <typename... TArgs>
02 auto sum(const TArgs&... args)
03 {
04     return (... + args);
05 }
06
07 auto result = sum(1, 2, 3, 4, 5);
```

Wyrażenie fold

## Składnia wyrażeń fold

Niech

$$e = e_1, e_2, \dots, e_n$$

będzie wyrażeniem, które zawiera nierozpakowany

*parameter pack* i



jest *operatorem fold*, wówczas **wyrażenie fold** ma postać

# Unary left fold

$$(\dots \otimes e)$$

```
01 template <typename... TArgs>
02 auto sum(const TArgs&... args)
03 {
04     return (... + args);
05 }
```

# Unary right fold

$$(e \otimes \dots)$$

```
01 template <typename... TArgs>
02 auto rsum(const TArgs&... args)
03 {
04     return (args + ...);
05 }
```

# Binary left fold

$$(a \otimes \dots \otimes e)$$

który jest rozwijany do postaci

$$(((a \otimes e_1) \dots) \otimes e_n)$$

```
01 template <typename... TArgs>
02 void print(const TArgs&... args)
03 {
04     (std::cout << ... << args);
05 }
```

# Binary right fold

$$(e \otimes \dots \otimes a)$$

który jest rozwijany do postaci

$$(e_1 \otimes (\dots (e_n \otimes a)))$$

```
01 template <typename... TArgs>
02 auto multiply(double factor, const TArgs&... args)
03 {
04     return (args * ... * factor);
05 }
```

# Operatory fold

- Operatorem  $\otimes$  może być jeden z poniższych operatorów C++:

```
01 + - * / % ^ & | ~ = < > << >>
02 += -= *= /= %= ^= &= |= <=< >=>
03 == != <= >= && || , .* ->*
```

# **std::string\_view**

# Klasa std::string\_view

- Nagłówek: <string\_view>
- Lekki uchwyt dla sekwencji znaków (read-only)
  - czas życia danych (bufora znaków) nie jest kontrolowany przez obiekt typu string\_view
  - brak wsparcia dla alokatorów - nie są potrzebne
  - przekazywanie przez wartość jest efektywne
  - typowa implementacja: wskaźnik const char\* i rozmiar
- Literał: sv
  - zdefiniowany w nagłówku
  - zdefiniowany jako constexpr

```
01 auto txt = "hello world"sv;
```

# Klasa std::string\_view - API

- Obiekt `string_view` zapewnia podobną funkcjonalność jak `std::string`:
  - `operator[]`
  - `at()`
  - `data()`
  - `size()`
  - `length()`
  - `find()`
  - `find_first_of()`
  - `find_last_of()`
- Zapewnia operatory porównania i wyliczania skrótu (`std::hash<std::string_view>`)

# Różnice między `string_view` a `string` (1)

- Wartość po konstrukcji domyślnej dla wewnętrznego wskaźnika to `nullptr`
  - `string::data()` nie może zwrócić `nullptr`

```
01 string_view txt;
02
03 assert(txt.data() == nullptr);
04 assert(txt.size() == 0);
```

## Różnice między `string_view` a `string` (2)

- Typ `string_view` nie ma gwarancji, że bufor znaków jest zakończony zerem (*null terminated string*)
  - Dla `string_view` zawsze należy sprawdzić rozmiar operacją `size()` zanim użyty zostanie operator`[]` lub wywołana zostanie metoda `data()`

```
01 char txt[3] = { 't', 'x', 't' };  
02  
03 string_view txt_v(txt, sizeof(txt)); // this view is not null terminated
```

## Konwersje **string** ↔ **string\_view**

```
01 std::string text = "abc def";  
02 std::string_view sv_text = text;
```

auto text = "abc def"sv; std::string str\_text{text};

## Konwersje **string** ↔ **string\_view**

```
01 std::string text = "abc def";  
02 std::string_view sv_text = text;
```

auto text = "abc def"sv; std::string str\_text{text};

- Konwersja **string** → **string\_view** jest szybka

# Konwersje **string** ↔ **string\_view**

```
01 std::string text = "abc def";  
02 std::string_view sv_text = text;
```

auto text = "abc def"sv; std::string str\_text{text};

- Konwersja **string** → **string\_view** jest szybka
  - dozwolona niejawnia konwersja przy pomocy `std::string::operator string_view()`

# Konwersje **string** ↔ **string\_view**

```
01 std::string text = "abc def";  
02 std::string_view sv_text = text;
```

auto text = "abc def"sv; std::string str\_text{text};

- Konwersja **string** → **string\_view** jest szybka
  - dozwolona niejawnia konwersja przy pomocy `std::string::operator string_view()`
- Konwersja **string\_view** → **string** jest kosztowna

# Konwersje **string** ↔ **string\_view**

```
01 std::string text = "abc def";  
02 std::string_view sv_text = text;
```

auto text = "abc def"sv; std::string str\_text{text};

- Konwersja **string** → **string\_view** jest szybka
  - dozwolona niejawnia konwersja przy pomocy `std::string::operator string_view()`
- Konwersja **string\_view** → **string** jest kosztowna
  - wymagana jawnia konwersja - `explicit std::string::string(string_view sv)`

# Użycie `string_view` w API funkcji

Obiekty `string_view` przekazywane jako argumenty wywołania funkcji powinny być przekazywane przez wartość:

```
01 void foo_s(const std::string& s);
02 void foo_sv(std::string_view sv);
03
04 foo_s("text"); // computes length, allocates memory, copies characters
05 foo_sv("text"); // computes only length
```

# Użycie `string_view` w API funkcji

- `string_view` powinno być stosowane zamiast `string` jeśli:
  - API nie wymaga, aby tekst był zakończony zerem
    - nie można przekazywać `string_view` do funkcji języka C!!!
  - odbiorca respektuje czas życia obiektu
  - dostęp do danych przy pomocy metody `data()` uwzględnia potencjalny pusty wskaźnik (`nullptr`)

# Użycie string\_view w API funkcji

- Należy unikać zwracania `string_view`, chyba że jest to świadomym wybór programisty
  - zwrócenie `string_view` może być niebezpieczne
  - należy pamiętać o tym, że `string_view` jest *non-owning view*

```
01 string_view start_from_word(string_view text, string_view word)
02 {
03     return text.substr(text.find(word));
04 }
```

# Użycie string\_view w API funkcji

- Należy unikać zwracania `string_view`, chyba że jest to świadomym wybór programisty
  - zwrócenie `string_view` może być niebezpieczne
  - należy pamiętać o tym, że `string_view` jest *non-owning view*

```
01 string_view start_from_word(string_view text, string_view word)
02 {
03     return text.substr(text.find(word));
04 }
```

- Jeśli wywołamy funkcję `start_from_word()` w następujący sposób, to mamy wiszący wskaźnik:

# Użycie string\_view w API funkcji

- Należy unikać zwracania `string_view`, chyba że jest to świadomym wybór programisty
  - zwrócenie `string_view` może być niebezpieczne
  - należy pamiętać o tym, że `string_view` jest *non-owning view*

```
01 string_view start_from_word(string_view text, string_view word)
02 {
03     return text.substr(text.find(word));
04 }
```

- Jeżeli wywołamy funkcję `start_from_word()` w następujący sposób, to mamy wiszący wskaźnik:

```
01 auto text = "one two three"s;
02 auto sv = start_from_word(text + " four", "two");
```

# Użycie string\_view w API funkcji

- Dostarczanie obydwu wersji funkcji jako przeciążeń może powodować dwuznaczności:

```
01 void foo(const string& s);  
02  
03 void foo(string_view sv);  
04  
05 foo("ambiguous"); // ERROR - ambiguous call
```

**std::any**

# Klasa std::any

Umożliwia:

- bezpieczne (typowane) mechanizmy przechowywania i odwoływania się do wartości dowolnych typów
  - bezpiecznie typizowany odpowiednik void\*
- przechowywanie elementów heterogenicznych w kontenerach biblioteki standardowej
- przekazywanie wartości dowolnych typów pomiędzy warstwami, bez konieczności wyposażania warstw pośredniczących w jakąkolwiek wiedzę o tych typach

# Wymagania

Typy przechowywane w `std::any` muszą:

- umożliwiać kopiowanie - typy move-only nie są wspierane, choć klasa
- umożliwiać przypisywanie (publiczny operator przypisania)
- nie mogą rzucać wyjątków z destruktora (to jest wymóg odnośnie wszystkich typów użytkownika w C++)

# Używanie std::any

```
01 std::any a;  
02  
03 a = std::string("Tekst...");  
04 a = 42;  
05 a = 3.1415;
```

```
01 double pi = std::any_cast<double>(a); // OK  
02  
03 std::string s = std::any_cast<std::string>(a); // throws std::bad_any_cast
```

```
01 Gadget* g = std::any_cast<Gadget>(&a); // zwraca nullptr  
02  
03 if (g)  
04     g->do_stuff();  
05 else  
06     std::cout << "Niepoprawna konwersja dla obiektu any.\n";
```

# Kontenery heterogeniczne & std::any

```
01 // predykat
02 std::vector<std::any> things = { 1, 3.14, "text"s, 42, 44.4f, 665 };
03 things.push_back(std::vector{1, 2, 3});
04 things.push_back(std::make_shared<Gadget>(42, "ipad"));
05
06 std::vector<std::any> numbers;
07
08 auto is_int = [] (const std::any& a) { return typeid(int) == a.type(); };
09 std::copy_if(things.begin(), things.end(), std::back_inserter(numbers), is
```

# **std::optional**

# **std::optional**

- Opcjonalnie przechowuje wartość określonego typu
  - jest pusty lub posiada określoną wartość
  - w rezultacie nie ma potrzeby korzystać ze specjalnych znaczników pustej wartości (np. NULL, -1, itp.)
- Nagłówek <optional>

## Tag pomocniczy - std::nullopt

- std::optional wykorzystuje stałą std::nullopt typu std::nullopt\_t jako specjalny znacznik oznaczający brak wartości dla obiektu

```
01 inline constexpr nullopt_t nullopt{ /*unspecified*/ };
```

# std::optional - konstrukcja obiektu

- w stanie be wartości:

```
01 std::optional<std::string> o1;  
02  
03 std::optional<double> o2 = std::nullopt;
```

- z określona wartością

```
01 std::optional<std::string> o3 = "text";  
02  
03 std::optional o4{42}; // deduces optional<int>
```

# std::optional - konstrukcja obiektu in-place

- na podstawie listy argumentów - bez konieczności tworzenia obiektu tymczasowego

```
01 std::optional<std::complex<double>> o5{std::in_place, 3.0, 4.0};  
02  
03 // initialize set with lambda as sorting criterion:  
04 auto sc = [] (int x, int y) {  
05     return std::abs(x) < std::abs(y);  
06 };  
07  
08 std::optional<std::set<int, decltype(sc)>> o6{std::in_place, {4, 8, -7, -2}}
```

## **std::make\_optional()**

```
01 auto o7 = std::make_optional(3.0); // optional<double>
```

# Sprawdzenie stanu

- Aby sprawdzić, czy obiekt opcjonalny przechowuje wartość możemy użyć:
  - metody `has_value()`
  - przeciążonej funkcji operator `bool`

```
01 std::optional o{42};  
02  
03 assert(o.has_value() == true);  
04  
05 if (o) // has value  
06 {  
07     //...  
08 }  
09  
10 if (!o) // is empty  
11 {  
12     //...  
13 }
```

# Dostęp do przechowywanej wartości - unsafe

```
01 std::optional opt_str = "text"s;  
02  
03 *opt_str = "other";  
04  
05 assert(opt_str.value() == "other");  
06 assert(opt_str->length() == 5);
```

- Użycie operatorów `*`, `*→` w sytuacji, gdy obiekt jest pusty (nie przechowuje wartości) skutkuje **undefined behavior**

# Dostęp do przechowywanej wartości - safe

- Bezpieczny dostęp do przechowywanej wartości może być zrealizowany poprzez metodę `const T& value()`
- Zwraca wartość. Jeśli jej nie ma rzuca wyjątkiem `std::bad_optional_access`

```
01 std::optional<std::string> opt_str;
02
03 try
04 {
05     string str = opt_str.value();
06 }
07 catch(const std::bad_optional_access& e)
08 {
09     //...
10 }
```

# Dostęp do przechowywanej wartości - safe

- T value\_or(U &&default\_value)
- Zwraca wartość lub jeśli jej nie ma, podaną jako argument wartość domyślną

```
01 std::optional<const char*> maybe_getenv(const char* n)
02 {
03     if(const char* x = std::getenv(n))
04         return x;
05     else
06         return std::nullopt;
07 }
08
09 //...
10 std::cout << maybe_getenv("MYPWD").value_or("(none)") << '\n';
```

# Resetowanie stanu

- Usunięcie wartości realizowane jest za pomocą metody `reset()`

```
01 std::optional password = "passwd";  
02  
03 password.reset();  
04  
05 assert(!password.has_value());
```

# std::optional - semantyka przenoszenia

- Klasa std::optional wspiera semantykę przenoszenia:

```
01 std::optional<std::string> os;
02
03 std::string text = "text";
04 os = std::move(text); // OK - string object is moved to optional
05
06 std::string destination = std::move(*os);
07
08 assert(os.has_value()); // os has value, but in unspecified state
```

## **std::optional - specjalne przypadki**

W przypadku zmiennych typu std::optional przechowywanie w nich wartości typu bool i wskaźników może mieć zaskakujące efekty

# std::optional

```
01 std::optional<bool> o{false};  
02  
03 if (!o) // yields false - o has value, which is false  
04 {  
05     //...  
06 }  
07  
08 if (o == false) // yields true  
09 {  
10     //...  
11 }
```

# std::optional

```
01 std::optional<double*> o{nullptr};  
02  
03 if (!o) // yields false - o has value  
04 {  
05     //...  
06 }  
07  
08 if (o == nullptr) // yields true  
09 {  
10     //...  
11 }
```

# Case study - opcjonalne składowe klasy

```
01 class Person
02 {
03     std::string first_name_;
04     std::optional<std::string> middle_name_;
05     std::string last_name_;
06 public:
07     Person(std::string fn, std::optional<std::string> mn, std::string ln)
08         : first_name_{std::move(fn)}, middle_name_{std::move(mn)}, last_name_{std::move(ln)}
09     {}
10
11     std::string full_name() const
12     {
13         return first_name_ + " " + (middle_name_ ? *middle_name_ + " " : "") + last_name_;
14     }
15 };
16
17 //...
18 Person p1{"Jan", "Maria", "Kowalski"};
19 assert(p1.full_name() == "Jan Maria Kowalski");
20
21 Person p2{"Jan", std::nullopt, "Kowalski"};
22 assert(p2.full_name() == "Jan Kowalski")
```

# **std::variant**

# Klasa std::variant

- umożliwia bezpieczne (ze względu na typy) przechowanie wartości określonego typu, wybranego z listy typów definiujących zmienną wariantową
  - jest implementacją koncepcji bezpiecznej unii (type-safe union)
- umożliwia statyczny podgląd (wizytację) zmiennych wariantowych
- efektywnie składuje wartości z listy typów wariantowych na stosie
- nie alokuje dynamicznie pamięci na stercie (istotna różnica w stosunku do std::any)
- nie może przechowywać referencji, tablic oraz typu void
- może zawierać duplikaty typów na liście
  - obsługa takiego przypadku jest realizowana poprzez indeksy (tak jak w std::tuple)

# Konstrukcja zmiennej wariantowej

- Deklarując typ wariantowy trzeba podać zestaw typów, które będą mogły być reprezentowane w typie wariantowym
- Domyślny konstruktor typu wariantowego inicjuje zmienną domyślną wartością dla pierwszego typu z listy

```
01 std::variant<int, string, double> my_variant1; // holds an int with a defa
02
03 std::variant<int, string, double> my_variant2(3.14); // holds a double 3.1
04
05 my_variant1 = 24;
06 my_variant1 = 2.52;
07 my_variant1 = "text"s;
```

# Monostate

- Jeśli pierwszy typ z listy nie jest domyślnie konstruowalny, można użyć tagu - `std::monostate`:

```
01 struct S
02 {
03     S(int v) : value{v}
04 }
05
06     int value;
07 };
08
09 std::variant<S, int> v1; // ERROR - ill-formed
10 std::variant<std::monostate, S, int> v2; // OK - now v2 must be assigned
```

# Przypisania wartości do zmiennej wariantowej

- operator=()

```
01 variant<int, string, double> v1;
02
03 v1 = 42; // v1 holds int{42}
04 v1 = "text"s; // v1 holds "text"s
05 v1 = 3.14; // v1 holds double{3.14}
06
07
08 variant<int, string, string> v2;
09 v2 = "text"s; // ERROR
10
11 variant<string, bool> v3;
12 v3 = "ctext"; // v3 holds bool{true}
```

# Przypisania wartości do zmiennej wariantowej

- `emplace(T&&...)`

```
01 class Gadget
02 {
03     int id_;
04     std::string name_;
05
06     Gadget(int id, std::string name);
07
08     //...
09 };
10
11 variant<int, Gadget, int> v;
12
13 v.emplace<Gadget>(1, "ipad"); // creates Gadget{1, "ipad"} inside variant object
14 v.emplace<0>(42); // sets the first int to 42
15 v.emplace<2>(665); // sets the second int to 665
```

# Przypisania wartości do zmiennej wariantowej

- `emplace(T&&...)`

```
01 class Gadget
02 {
03     int id_;
04     std::string name_;
05
06     Gadget(int id, std::string name);
07
08     //...
09 };
10
11 variant<int, Gadget, int> v;
12
13 v.emplace<Gadget>(1, "ipad"); // creates Gadget{1, "ipad"} inside variant object
14 v.emplace<0>(42); // sets the first int to 42
15 v.emplace<2>(665); // sets the second int to 665
```

Gdy na liście typów występują duplikaty

# Dostęp do wartości wariantowej

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`
- Funkcja `T* std::get_if<T>(v)`

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`
- Funkcja `T* std::get_if<T>(v)`
  - w razie niezgodności typów, zwrócony zostanie `nullptr`

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`
- Funkcja `T* std::get_if<T>(v)`
  - w razie niezgodności typów, zwrócony zostanie `nullptr`

```
01 std::variant<int, std::string, double> my_variant{"text"s};  
02  
03 std::string s1 = std::get<std::string>(my_variant); // OK  
04 std::string s2 = std::get<1>(my_variant); // OK  
05 std::get<string>(v) += "!!!!";  
06  
07 int x = std::get<int>(my_variant); // ERROR - throws std::bad_variant_access  
08  
09 if (std::string* ptr_str = std::get_if<std::string>(&my_variant); ptr_str != nullptr)  
10     cout << "Stored string: " << *ptr_str << endl;
```

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`
- Funkcja `T* std::get_if<T>(v)`
  - w razie niezgodności typów, zwrócony zostanie `nullptr`

```
01 std::variant<int, std::string, double> my_variant{"text"s};  
02  
03 std::string s1 = std::get<std::string>(my_variant); // OK  
04 std::string s2 = std::get<1>(my_variant); // OK  
05 std::get<string>(v) += "!!!!";  
06  
07 int x = std::get<int>(my_variant); // ERROR - throws std::bad_variant_access  
08  
09 if (std::string* ptr_str = std::get_if<std::string>(&my_variant); ptr_str != nullptr)  
10     cout << "Stored string: " << *ptr_str << endl;
```

# Dostęp do wartości wariantowej

- Funkcja `T& std::get<T>(v)` lub `T& std::get<Index>(v)`
  - jeśli wywołanie `std::get(v)` okaże się nieskuteczne (niezgodne typy), zgłoszany jest wyjątek `std::bad_variant_access`
- Funkcja `T* std::get_if<T>(v)`
  - w razie niezgodności typów, zwrócony zostanie `nullptr`

```
01 std::variant<int, std::string, double> my_variant{"text"s};  
02  
03 std::string s1 = std::get<std::string>(my_variant); // OK  
04 std::string s2 = std::get<1>(my_variant); // OK  
05 std::get<string>(v) += "!!!!";  
06  
07 int x = std::get<int>(my_variant); // ERROR - throws std::bad_variant_access  
08  
09 if (std::string* ptr_str = std::get_if<std::string>(&my_variant); ptr_str != nullptr)  
10     cout << "Stored string: " << *ptr_str << endl;
```

# holds\_alternative & index

- `bool holds_alternative<T>(const std::variant &v)`
  - Sprawdza, czy zmienna wariantowa przechowuje w danym momencie odpowiedni typ

```
01 if (std::holds_alternative<double>(v))  
02     std::cout << "Holds double\n";
```

- `std::size_t index() const`
  - Zwraca indeks (licząc od zera) typu z listy dla danego stanu zmiennej wariantowej

```
01 std::variant<int, double> v = 3.14;  
02 assert(v.index() == 1);
```

# Problem pustego stanu

- Obiekt klasy `std::variant` może stać się pustym obiektem tylko w przypadku wystąpienia wyjątku w trakcie operacji przypisania nowej (lub domyślnej) wartości.
- W takim przypadku:
  - metoda `valueless_by_exception()` zwraca `true`
  - a wywołanie metody `index()` zwraca wartość `std::variant_npos`

# Problem pustego stanu

```
01 struct S
02 {
03     operator int()
04     {
05         throw std::runtime_error("ERROR#13");
06     }
07 };
08
09 std::variant<int, double> v{3.14};
10
11 try
12 {
13     v.emplace<0>(S{}); // throws while being set
14 }
15 catch(...)
16 {
17     assert(v.valueless_by_exception() == true);
```

# Wizytowanie wariantów

- Rozwiązaniem problemu przeglądania wartości przechowywanych w zmiennych wariantowych jest zastosowanie **wizytatora**
- Wizytatory są implementowane jako obiekty funkcyjne z operatorami wywołania funkcji przyjmującymi argumenty typów odpowiadających typom z zestawu wariantowego
- Wizytacja odbywa się za pośrednictwem funkcji `std::visit(wizytator, zmienna-wariantowa)`
- Jeżeli zmieni się zestaw typów w zmiennej wariantowej, która była wizytowana przez wizytatora i wizytator nie będzie w stanie obsłużyć nowo dodanego typu, to kompilator zgłosi błąd

# Klasa wizytatora

```
01 class PrintVisitor
02 {
03 public:
04     void operator()(int i) const
05     {
06         cout << "int: " << i << "\n";
07     }
08
09     void operator()(string s) const
10     {
11         cout << "string: " << s << "\n";
12     }
13 };
14
15 std::variant<int, string> var("Test"s);
16
17 PrintVisitor pv;
18 std::visit(pv, var); // compile error if type is not supported by visitor
19
20 var = 12;
21 std::visit(PrintVisitor{}, var);
```

# Wizytacja za pomocą lambd

- Do wizytacji można również wykorzystać lambdę generyczną:

```
01 std::visit([](auto&& value) { std::cout << value << "\n" }, var);
```

# [] + [] == overload

- Istnieje możliwość zbudowania wizytora, składającego się z obiektów domknieć w miejscu wizytacji (in-place)
- Zbudowanie typu zawierającego zbiór lambd (*overload resolution set*) polega na dziedziczeniu po klasach domknieć

```
01 template <typename... Ts>
02 struct overload : Ts...
03 {
04     using Ts::operator()...
05 };
06
07 template <typename... Ts>
08 overload(Ts...) → overload<Ts...>;
09
10 auto local_visitor = overload {
11     [](int value) { return "int: "s + to_string(value); },
12     [](double value) { return "double: "s + to_string(value); },
13     [](const string s) { return "string: " + s; }
14 };
```

# overload & wizytacja

```
01 std::variant<int, double, string> v = 42;
02
03 auto local_visitor = overload {
04     [](int value) { return "int: "s + to_string(value); },
05     [](double value) { return "double: "s + to_string(value); },
06     [](const string s) { return "string: " + s; }
07 };
08
09 auto result = visit(local_visitor, v);
10 assert(result == "int: 42"s);
11
12 v = text;
13 result = visit(local_visitor, v);
14 assert(result == "string: text");
```

# Parallel STL

# Parallel STL

- W C++17 istnieje możliwość współbieżnego wykonania algorytmów STL
- Większość znanych algorytmów posiada nową wersję umożliwiającą wykonanie algorytmu współbieżnie
- Dodano też nowe algorytmy:
  - `reduce()`
  - `transform_reduce()`
  - `transform_exclusive_scan` & `transform_inclusive_scan`
- Bazą dla wprowadzenia współbieżności do STL był dokument Parallel TS

# Funkcje dostępu do danych

- Algorytmy współbieżne korzystają z tzw. funkcji dostępu do danych (element access functions). \* Należą do nich:
  - wszystkie operacje danej kategorii iteratora przekazanego przy wywołaniu algorytmu
  - operacje wykonywane na elementach wymagane przez specyfikację algorytmu
  - obiekty funkcyjne przekazane przez użytkownika oraz operacje na tych obiektach wymagane przez specyfikację

# Funkcje dostępu do danych

- Na przykład algorytm `sort()`:
  - wykorzystuje operacje iteratora o dostępie swobodnym
  - wywołuje funkcję `swap()` na elementach sekwencji
  - wywołuje funktor `Compare` przekazany przez użytkownika

## Funkcje lub obiekty funkcyjne

przekazane do algorytmu współbieżnego nie powinny bezpośrednio lub pośrednio modyfikować wartości obiektów przekazanych do nich jako argumenty

# Wytyczne wykonania algorytmów współbieżnych

- Umożliwiają żądanie wykonania algorytmu we współbieżny lub sekwencyjny sposób
- Jest to realizowane poprzez przekazanie jako pierwszego argumentu wywołania funkcji tzw. wytycznej wykonania (execution policy)

# **std::execution::seq**

- Typ `std::execution::sequenced_policy`
- Jeden wątek (ten, w którym został wywołany algorytm) wykonuje wszystkie zadania sekwencyjnie w pewnej kolejności, która nie jest ścisłe zdefiniowana i może być z każdym wywołaniem inna
  - Nie ma gwarancji, że kolejność wykonywania operacji będzie taka sama jak w wersji algorytmu bez wytycznej

# std::execution::seq

```
01 std::vector<int> v(1000);
02 int count = 0;
03
04 std::for_each(std::execution::seq, v.begin(), v.end(),
05               [&](int& x){ x = ++count; }); // not a race because of std::
```

# std::execution::seq

- Tryb wykonania przydatny przy debugowaniu i w testach
- Brak współbieżności wykonania - nie ma konieczności synchronizacji dostępu do współdzielonych danych

```
01 std::ofstream log_file("log.dat");
02
03 std::transform(std::execution::seq, data.begin(), data.end(), dest.begin()
04                 [&log_file](auto x) {
05                     log_file << x; // safe - no concurrency, no threads
06                     return x * x;
07                 }
08 );
```

# **std::execution::par**

- Typ: `execution::parallel_policy`
- Wiele wątków może wykonywać współbieżnie zadania (multithreading)
- Zadania w obrębie swojego wątku roboczego są wykonywane sekwencyjnie w zadanej (lecz nieokreślonej) kolejności, bez przeplotu (not-interleaved) => wszystkie zadania muszą być thread safe
- Należy zapewnić, że operacje wykonywane przez algorytm są wolne od data-race i nie powodują deadlock'ów
- Istnieje możliwość użycia konstrukcji synchronizujących współbieżny dostęp do danych (np. `std::mutex`, `std::atomic<T>`)

## **std::execution::par**

Aby nie dopuścić do niezdefiniowanego zachowania programu (UB) musimy zsynchronizować dostęp do współdzielonych zasobów:

# std::execution::par

- stosując zmienną typu atomowego

```
01 std::vector<int> v(1000);
02 std::atomic<int> count{};
03
04 std::for_each(
05     std::execution::par, v.begin(), v.end(),
06     [&](int& x){ x = ++count; }
07 ); // must be atomic when std::execution::par
```

# std::execution::par

- stosując mutex

```
01 std::transform(             
02     std::execution::par, data.begin(), data.end(), dest.begin(),  
03     [&log_file](auto x) {  
04         {  
05             std::lock_guard lk{log_file_mutex};  
06             log_file << x; // now concurrent access is synchronized  
07         }  
08         return x * x;  
09     }  
10 );
```

# **std::execution::par\_unseq**

- Typ: `execution::parallel_unsequenced_policy`
- Zadania mogą być wykonywane z wykorzystaniem wielowątkowości (multithreading) i współbieżności wektorowej (np. OpenMP)
- Zadania mogą być:
  - wykonywane w różnej kolejności w różnych wątkach
  - przemieszane (*interleaved*) w ramach konkretnego wątku (np. druga operacja zostanie rozpoczęta zanim pierwsza zostanie ukończona)
  - transferowane między wątkami (zadanie rozpoczęte w wątku nr 1, może kontynuować pracę w wątku nr 2 i zakończyć działanie w wątku nr 3)

## **std::execution::par\_unseq**

- Wywołania operacji synchronizujących (np. mutex::lock()) grożą zakleszczeniem
- Nie można używać dynamicznej alokacji i dealokacji pamięci
- Operacje (funkcje) wykonywane przez algorytm muszą operować tylko na zadanym elemencie kolekcji i nie mogą modyfikować jakiegokolwiek współdzielonego stanu pomiędzy wątkami lub elementami sekwencji

# std::execution::par\_unseq - możliwy deadlock

```
01 int x = 0;
02 std::mutex m;
03 int a[] = {1,2};
04
05 std::for_each(
06     std::execution::par_unseq, std::begin(a), std::end(a),
07     [&](int) {
08         std::lock_guard lk(m); // Error: lock_guard constructor calls m.lo
09         ++x;
10     }
11 );
```

# std::execution::par\_unseq - OK

```
01 std::transform(02     std::execution::par_unseq, data.begin(), data.end(), dest.begin(),03     [&](auto x) {04         return x * x; // OK - no access to a shared state05     }06 );
```

## Wyjątki w algorytmach współbieżnych

Jeśli jakkolwiek wyjątek wydostanie się z algorytmu współbieżnego, wywołana zostanie funkcja terminate()

## **std::reduce**

- Działa jak algorytm `std::accumulate()`, ale aplikuje funktor binarny w nieokreślonej kolejności
- Domyślnym funktorem jest `std::plus<>`
- Rezultat w przypadku przekazania funktora, który nie jest przechodni i komutatywny, jest nieokreślony
  - np: dodawanie zmiennych typu `float`

# std::reduce

```
01 std::vector<int> v(1000665);
02 std::iota(v.begin(), v.end(), 1);
03
04 auto sum = std::reduce(std::execution::par_unseq, v.begin(), v.end(), 0LL)
```

## **std::transform\_reduce**

- Aplikuje transformację funktem unarnym lub binarnym, a następnie redukuje wyniki transformacji funktem binarnym
- Domyślnymi funktorami są odpowiednio: std::multiplies<> i std::plus<>

# std::transform\_reduce

```
01 std::uintmax_t count_words(string_view text)
02 {
03     if (text.empty())
04         return 0;
05
06     auto is_word_beginning = [](auto left, auto right) {
07         return std::isspace(left) && !std::isspace(right);
08     };
09
10    std::uintmax_t wc = (!std::isspace(text.front()) ? 1 : 0);
11
12    wc += std::transform_reduce(
13        std::par_unseq,
14        text.begin(), text.end() - 1,
15        text.begin() + 1,
16        0ULL,
17        std::plus<>(),
18        is_word_beginning);
19
20    return wc;
21 }
```