Szablony

Problem

Jak stworzyć funkcję, która zwraca mniejszą z dwóch liczb?

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

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 funkcji

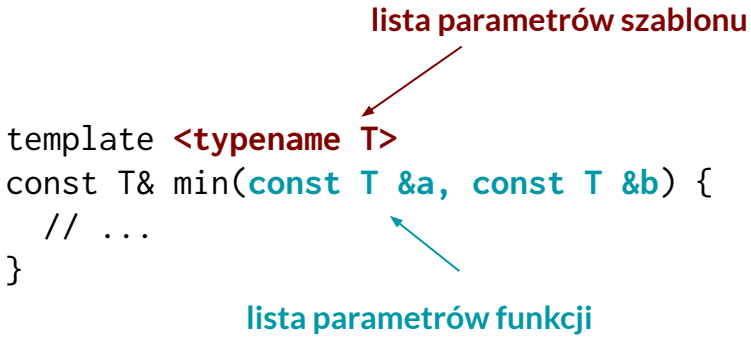
- wzorzec klasy, funkcji lub zmiennej, który może zostać wykorzystany przez kompilator do utworzenia prawdziwej klasy, funkcji lub zmiennej
- parametryzowany typami, wartościami lub szablonami

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

Szablony funkcji

- 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



Szablony funkcji

- szablon może mieć dowolną liczbę parametrów szablonowych

```
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 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};
```

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

Szablony bez szablonów - C++20 + auto

```
auto min(auto lhs, auto rhs) {
        return lhs < rhs ? lhs : rhs;
}
```

Biblioteka standardowa, programowanie generyczne

Składowe języka C++

- core language features
  - elementy, które wymagają zmiany składni języka
- biblioteka standardowa
  - elementy, które da się stworzyć poprzez zbudowanie z istniejących elementów

Typy danych

- pair
- tuple
- kolekcje
  - vector
  - deque
  - map
  - unordered_map
  - set
  - unordered_set
  - list

vector - pojemnik pierwszego wyboru

- w nowoczesnych komputerach `vector` będzie średnio najszybszy
  - elementy sąsiadująco w pamięci
  - https://quick-bench.com/q/IPNBidd8q5jajdSQoaWvx44r_nQ
- najważniejsze aspekty
  - rośnie “od ogona”
    - dopychanie na koniec względnie tanie
    - dopychanie na początek (lub w środku) względnie drogie
    - warto wołać `reserve()`, jeżeli znamy liczbę elementów, której potrzebujemy
  - wydajność mocno zależy od charakterystyki przechowywanego typu
    - dodanie elementu potencjalnie dokonuje realokację pamięci
    - istniejące elementy są memcpy/przesuwane/kopiowane na nowe miejsce
    - bardzo łatwo popsuć wydajność - https://quick-bench.com/q/5Cw9DIGobK4MnWlJdm6Tz7HCv7E

vector - emplace_back vs push_back

- push_back, insert
  - podajemy element, który zostanie dodany do wektora
  - element tworzony jest zanim zostanie przekazany (kopia/przeniesienie) do wektora
- emplace_back, emplace
  - podajemy argumenty wywołania konstruktora elementu, który ma być dodany
  - element tworzony jest już w wektorze (nie ma kopii ani przeniesienia)
  - zwraca referencję na element w wektorze

span (C++20)

- Jak napisać funkcję, co pobiera zestaw elementów sąsiadujących w pamięci?
  - char tab[], const char*, std::string, std::string_view, std::vector<std::string>, …?
- wszystkie te typy to w zasadzie wskaźnik + rozmiar
  - void foo(char *ptr, size_t N)
- łatwo zapomnieć/przegapić/pomylić się w podaniu rozmiaru i będzie kłopot
  - wynika to z przekazywania oddzielnie wskaźnika i rozmiaru
  - a gdyby one były scalone w jeden obiekt?
- std::span
  - void foo(std::span<char> str);
  - automatyczna konwersja z kompatybilnych typów
- dynamiczny (std::span<int>) vs statyczny (std::span<int, N>) rozmiar

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;
  }
}
```

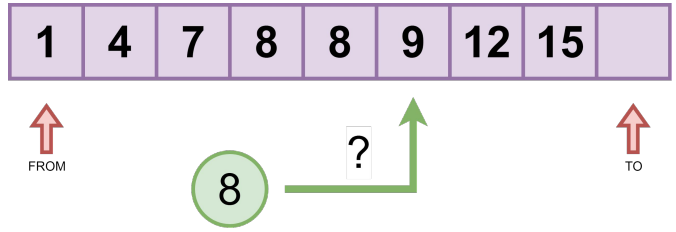
```
void selection_sort(auto from, auto to)
{
   for (; from != to; ++from) {
      auto iter = std::min_element(from, to);
      std::iter_swap(from, iter);
   }
}
```

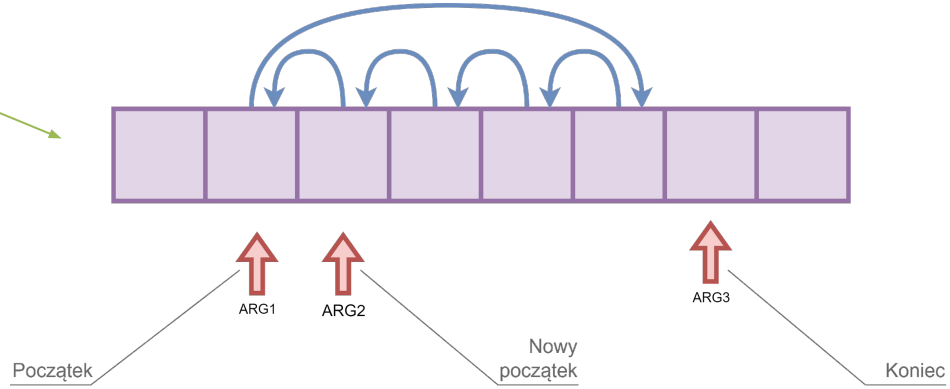
Co to jest za algorytm?

```
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

```
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

- Std C++ posiada ponad 100 algorytmów ogólnego przeznaczenia
- w C++20 dochodzą wersje części algorytmów w postaci uzakresowionej
- poza algorytmami jest mnóstwo funkcji pomocniczych
- algorytmy opierają się na iteratorach

Zalety używania gotowych algorytmów

- czytelność
  - dobrze nazwane funkcje zamiast ściany kodu
  - Clean Code → funkcja wołająca działa na jednym poziomie abstrakcji
- odporność na błędy
  - przetestowane przez miliony użytkowników w ciągu kilkudziesięciu lat
  - automatycznie dostajemy ewentualne poprawki przy rekompilacji z nowszą STD
- wydajność
  - algorytmy dopasowują się do charakterystyki danych, na których operują
  - kompilator “widzi” kod (szablony), co pozwala mu go zoptymalizować

https://www.youtube.com/watch?v=W2tWOdzgXHA (“No raw loops”, Sean Parent)

Iterator

- 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 wgłąb) bez modyfikacji interfejsu agregatu

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

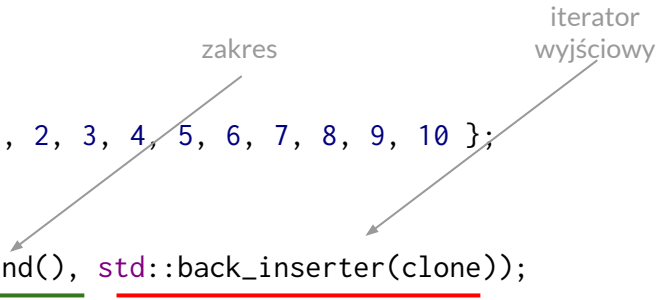
```cpp
#include <algorithm>
#include <iterator>
#include <iostream>
#include <vector>

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

    std::copy(v.begin(), v.end(), std::back_inserter(clone));

    std::copy(v2.begin(), v2.end(), std::ostream_iterator<int>(std::cout, " ")); std::cout << '\n';

    return 0;
}
```



Algorytmy z modyfikatorami - kopiowanie warunkowe

```
bool filter(int value) {
    return value > 0 && value < 10;
}
```

```
const std::set<int> s = {1, 5, 3, 8, 9, 15, 14, 0};
std::array<int, 8> result{};
std::copy_if(s.begin(), s.end(), result.begin(), filter);
```

Algorytmy z modyfikatorami - kopiowanie warunkowe

```
const std::set<int> s = {1, 5, 3, 8, 9, 15, 14, 0};
std::array<int, 8> result{};
std::copy_if(s.begin(), s.end(),
             result.begin(), [](int value) { return value > 0 && value < 10; }
);
```

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

```
int count_less_than(auto from, auto to, int threshold) {
   auto predicate = [threshold] (int value) { return value < threshold; };
   return std::count_if(from, to, predicate);
}
```

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

- 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

```
[](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 zawołać 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;
};
```

Algorytmy - sprawdzanie

- any_of
- all_of
- none_of

Algorytmy - wyszukiwanie

- find
- find_if
- adjacent_find
- mismatch
- search

Algorytmy - wyszukiwanie binarne

- lower_bound
- upper_bound
- binary_search
- equal_range

Algorytmy - zliczanie

- count
- count_if

Algorytmy - modyfikacja

- copy
- copy_if
- copy_n
- copy_backward
- move
- move_backward
- fill
- fill_n
- transform
- generate
- generate_n
- replace
- replace_if
- reverse
- rotate
- unique

Algorytmy - numeryczne

- iota
- accumulate
- inner_product
- partial_sum

Algorytmy - usuwanie

- remove
- remove_if

Algorytmy - losowość

- generatory losowości
  - random_device
  - mt19937
  - rozkłady zmiennych losowych
- algorytmy
  - shuffle
  - sample

Algorytmy

… i wiele innych

https://en.cppreference.com/w/cpp/algorithm

https://en.cppreference.com/w/cpp/numeric

https://www.youtube.com/watch?v=2olsGf6JIkU
(105 STL algorithms in less than an hour)

Algorytmy - wersje uzakresowione

- Od C++20 możemy używać zakresów (ranges)
- dwa oddzielne obiekty iteratorów zastąpione są jednym (iterator+strażnik)
- daje to całkiem nowe możliwości, o których tu nie mamy czasu powiedzieć…
- … ale daje też nowe wersje algorytmów
  - zakres zamiast pary iteratorów → `std::ranges::sort(v);`
  - odwzorowania (projections) → `std::ranges::find(people, “Piotr”, &Person::name);`

Pozostałe

- biblioteka standardowa to nie tylko kontenery i algorytmy
  - funkcje pomocnicze
  - wielowątkowość
  - czas i kalendarz
  - metaprogramowanie
  - formatowanie tekstu
  - liczby zespolone
  - system plików
  - wyrażenia regularne

Programowanie uogólnione

- Tworzenie kodu programu
  - bez znajomości typów danych
  - zachowujące kontrolę typów
- W skrócie
  - używamy typów wg tego co potrafią, a nie jakimi konkretnie typami są i po czym dziedziczą

- Realizacja
  - polimorfizm statyczny
  - span
  - iteratory
  - auto

Na czym polega polimorfizm?

Polimorfizm

- przeciążanie funkcji
- szablony
- warianty
- polimorfizm dynamiczny (funkcje wirtualne)

Dedukcja typu zmiennej

```
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 wyniku funkcji

`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);
}
```

Almost-Always-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 };
```