

PRZEDMIOT: Elementy programowania

KLASA: 1i gr. 2

Lekcja

Temat: Szablony (templates): Szablony funkcji i klas, specjalizacje, szablony w STL.

1 Szablony (templates) w C++

Szablony pozwalają pisać **uniwersalny kod**, który działa dla różnych typów danych **bez powielania kodu**. Czyli "napisz raz, a zastosuj dla wielu typów"

2 Szablony funkcji

✓ Składnia ogólna:

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

- **T** - to **zmienna typu**, którą kompilator zastąpi konkretnym typem w momencie użycia funkcji lub klasy. Dzięki temu możesz pisać **jedną funkcję lub klasę**, która działa dla wielu typów danych.
- **template <typename T>** - deklaruje szablon z typem T
- Funkcja **maksimum** działa teraz dla **int, double, float, itp.**

✓ Przykład użycia:

```
#include <iostream>
using namespace std;

template <typename T>
T maksimum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maksimum(5, 10) << endl;      // int
    cout << maksimum(3.14, 2.71) << endl; // double
}
```

3 Szablony klas

Szablony działają też dla **klas** — pozwalają tworzyć klasy działające na różnych typach danych.

```
#include <iostream>
using namespace std;
template <typename T>
class Para {
    T pierwszy, drugi;
public:
    Para(T a, T b) : pierwszy(a), drugi(b) {}
    T suma() { return pierwszy + drugi; }
};

int main() {
    Para<int> p1(3, 7);
    cout << p1.suma() << endl; // 10

    Para<double> p2(2.5, 3.5);
    cout << p2.suma() << endl; // 6.0
}
```

4 Specjalizacje szablonów

Czasem chcemy, aby szablon działał **inaczej dla konkretnego typu**.

```
#include <iostream>
using namespace std;

// Szablon klasy ogólny
template <typename T>
class Para {
    T pierwszy, drugi;
public:
    Para(T a, T b) : pierwszy(a), drugi(b) {}
    T suma() { return pierwszy + drugi; }
    void pokaz() { cout << pierwszy << " " << drugi << endl; }
};

// Specjalizacja szablonu dla typu char
template <>
class Para<char> {
    char pierwszy, drugi;
public:
```

```

Para(char a, char b) : pierwszy(a), drugi(b) {}
void pokaz() {
    cout << "Specjalizacja dla char: " << pierwszy << " " << drugi << endl;
}
};

int main() {
    // Użycie szablonu ogólnego dla int
    Para<int> p1(3, 7);
    cout << "Suma int: " << p1.suma() << endl;
    p1.pokaz();

    // Użycie szablonu ogólnego dla double
    Para<double> p2(2.5, 3.5);
    cout << "Suma double: " << p2.suma() << endl;
    p2.pokaz();

    // Użycie specjalizacji dla char
    Para<char> p3('A', 'B');
    p3.pokaz();

    return 0;
}

```

- To nazywamy **pełną specjalizacją**.
- Możemy też tworzyć **częściowe specjalizacje**, np. dla wskaźników.

5 Szablony w STL (Standard Template Library)

STL to **biblioteka standardowa w C++**, która w dużej mierze **opiera się na szablonach**.

Przykłady:

1. **vector** — dynamiczna tablica

```

#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v = {1, 2, 3};
    v.push_back(4);

    for (int x : v)

```

```
        cout << x << " ";
}
```

2. **map** — mapa klucz-wartość

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
    map<string, int> wiek;
    wiek["Jan"] = 25;
    wiek["Anna"] = 30;

    for (auto &[k, v] : wiek)
        cout << k << " ma " << v << " lat" << endl;
}
```

3. **sort** w **<algorithm>** — szablon funkcji

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v = {3, 1, 4, 2};
    sort(v.begin(), v.end()); // sort działa dla każdego typu porównywalnego
    for (int x : v) cout << x << " ";
}
```

W STL wszystko jest napisane przy użyciu **szablonów**, dlatego możesz używać **vector<int>**, **vector<double>**, **map<string,int>** itd., bez pisania osobnej klasy.

Lekcja

Temat: STL (Standard Template Library) w C++

STL (Standard Template Library) to jedna z najważniejszych części języka C++.

Zawiera:

- **kontenery** — struktury danych (vector, list, map, queue itd.)
- **iteratory** — „wskaźniki” do elementów kontenerów
- **algorytmy** — sortowanie, wyszukiwanie, kopiowanie itd.

1 vector — dynamiczna tablica

vector to **dynamiczna tablica**, która sama zmienia rozmiar podczas dodawania/usuwania elementów.

📌 Zastosowania

- przechowywanie listy liczb
- zbiór danych od użytkownika
- tablica, której rozmiar nie jest znany

📌 Przykład

```
vector<int> liczby;
liczby.push_back(10);
liczby.push_back(20);
liczby.push_back(30);
```

📌 Iteracja po vectorze za pomocą iteratora

```
for (vector<int>::iterator it = liczby.begin(); it != liczby.end(); ++it) {
    cout << *it << " ";
}
```

Przykład:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v;

    // --- DODAWANIE NA KONCU ---
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
```

```

std::cout << "Po dodaniu na koncu (push_back): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

// --- USUWANIE Z KOŃCA ---
v.pop_back();
std::cout << "Po usunieciu z konca (pop_back): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

// --- DODAWANIE NA POCZĄTKU ---
v.insert(v.begin(), 5);

std::cout << "Po dodaniu na poczatku (insert): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

// --- USUWANIE Z POCZĄTKU ---
v.erase(v.begin());

std::cout << "Po usunieciu z poczatku (erase): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

// --- DODAWANIE W DOWOLNE MIEJSCE ---
v.insert(v.begin() + 1, 99);

std::cout << "Po dodaniu w dowolne miejsce (index 1): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

// --- USUWANIE Z DOWOLNEGO MIEJSCA ---
v.erase(v.begin() + 1);

std::cout << "Po usunieciu z dowolnego miejsca (index 1): ";
for (int x : v) std::cout << x << " ";
std::cout << "\n";

return 0;
}

```

list — lista dwukierunkowa

list to **lista dwukierunkowa (double linked list)**.

Wstawianie i usuwanie elementów jest **bardzo szybkie**, ale dostęp po indeksie jest wolny.

Zastosowania

- kolejki z częstym dodawaniem i usuwaniem
- implementacja historii działań (przód/tył)
- struktur danych, gdzie ważny jest szybki insert w środku

Przykład

```
list<string> slowa;
slowa.push_back("Ala");
slowa.push_back("ma");
slowa.push_back("kota");
```

Iterator

```
for (list<string>::iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << " ";
}
```

Przykład

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst;

    // --- DODAWANIE NA KOŃCU ---
    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);

    std::cout << "Po dodaniu na koncu (push_back): ";
    for (int x : lst) std::cout << x << " ";
    std::cout << "\n";

    // --- USUWANIE Z KOŃCA ---
    lst.pop_back();

    std::cout << "Po usunieciu z konca (pop_back): ";
    for (int x : lst) std::cout << x << " ";
    std::cout << "\n";

    // --- DODAWANIE NA POCZĄTKU ---
    lst.push_front(5);

    std::cout << "Po dodaniu na poczatku (push_front): ";
    for (int x : lst) std::cout << x << " ";
    std::cout << "\n";

    // --- USUWANIE Z POCZĄTKU ---
    lst.pop_front();

    std::cout << "Po usunieciu z poczatku (pop_front): ";
    for (int x : lst) std::cout << x << " ";
```

```

std::cout << "\n";

// --- DODAWANIE W DOWOLNE MIEJSCE ---
auto it = lst.begin();
std::advance(it, 1);
lst.insert(it, 99);

std::cout << "Po dodaniu w dowolne miejsce (index 1): ";
for (int x : lst) std::cout << x << " ";
std::cout << "\n";

// --- USUWANIE Z DOWOLNEGO MIEJSCA ---
it = lst.begin();
std::advance(it, 1);
lst.erase(it);

std::cout << "Po usunięciu z dowolnego miejsca (index 1): ";
for (int x : lst) std::cout << x << " ";
std::cout << "\n";

return 0;
}

```

🔥 Budowa wewnętrzna

vector

- przechowuje elementy **w jednej ciągłej tablicy w pamięci**
- indeksowanie działa jak w tablicy: v[0], v[1], ...

list

- to **lista dwukierunkowa** – każdy element zawiera wskaźniki do poprzedniego i następnego
- elementy **są porozrzucane w pamięci**

🔥 Prosty przykład porównawczy

vector

```

std::vector<int> v;
v.push_back(10);
v[0] = 5; // szybki dostęp

```

list

```

std::list<int> lst;
lst.push_back(10);

auto it = lst.begin();
*it = 5; // można tylko przez iterator

```

3 map — klucz → wartość (słownik)

`map` przechowuje pary **klucz–wartość**
(Klucze są automatycznie sortowane!).

📌 **Zastosowania**

- słowniki (np. "PL" → "Polska")
- dane użytkowników (ID → nazwa)
- liczenie wystąpień słów

📌 **Przykład**

```
map<string, int> wiek;
wiek["Adam"] = 25;
wiek["Beata"] = 30;
wiek["Czarek"] = 22;
```

📌 **Iterator**

```
for (map<string, int>::iterator it = wiek.begin(); it != wiek.end(); ++it) {
    cout << it->first << " ma " << it->second << " lat\n";
}
```

Poniższy przykład zawiera:

- ✓ dodawanie elementów
- ✓ usuwanie elementu o najmniejszym kluczu (początek mapy)
- ✓ usuwanie elementu o największym kluczu (koniec mapy)
- ✓ usuwanie/dodawanie elementów według klucza (czyli "w dowolnym miejscu")

```
#include <iostream>
#include <map>

int main() {
    std::map<int, std::string> mp;

    // --- DODAWANIE ELEMENTÓW ---
    mp.insert({2, "B"});
    mp.insert({1, "A"});
    mp.insert({3, "C"});

    std::cout << "Po dodaniu elementow:\n";
    for (auto &p : mp) {
        std::cout << p.first << " -> " << p.second << "\n";
    }

    // --- USUWANIE NAJMIEJSZEGO KLUCZA (początek) ---
    mp.erase(mp.begin());

    std::cout << "\nPo usunięciu najmniejszego klucza:\n";
    for (auto &p : mp) {
        std::cout << p.first << " -> " << p.second << "\n";
    }

    // --- USUWANIE NAJWIEKSZEGO KLUCZA (koniec) ---
    auto it = mp.end();
```

```

it--; // ostatni element
mp.erase(it);

std::cout << "\nPo usunieciu najwiekszego klucza:\n";
for (auto &p : mp) {
    std::cout << p.first << " -> " << p.second << "\n";
}

// --- DODAWANIE "W DOWOLNE MIEJSCE" (przez klucz) ---
mp[10] = "X";
mp[5] = "Y"; // automatycznie trafi w odpowiednie miejsce

std::cout << "\nPo dodaniu elementow o kluczach 10 i 5:\n";
for (auto &p : mp) {
    std::cout << p.first << " -> " << p.second << "\n";
}

// --- USUWANIE ELEMENTU O DOWOLNYM KLUCZU ---
mp.erase(5);

std::cout << "\nPo usunieciu klucza 5:\n";
for (auto &p : mp) {
    std::cout << p.first << " -> " << p.second << "\n";
}

return 0;
}

```

🔥 Kompletny przykład

Program demonstruje użycie:

- **vector**
- **list**
- **map**
- **iteratorów**

```

#include <iostream>
#include <vector>
#include <list>
#include <map>
using namespace std;

int main() {

    // --- VECTOR ---
    vector<int> v = {1, 2, 3, 4, 5};

    cout << "VECTOR: ";
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";
    }
    cout << "\n";
}

```

```

// --- LIST ---
list<string> l;
l.push_back("Ala");
l.push_back("ma");
l.push_back("kota");

cout << "LIST: ";
for (list<string>::iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << " ";
}
cout << "\n";

// --- MAP ---
map<string, int> wiek;
wiek["Adam"] = 25;
wiek["Beata"] = 30;
wiek["Czarek"] = 22;

cout << "MAP:\n";
for (map<string, int>::iterator it = wiek.begin(); it != wiek.end(); ++it) {
    cout << it->first << " ma " << it->second << " lat\n";
}

return 0;
}

```

👉 v.begin()

Zwraca iterator **na pierwszy element** wektora.

👉 v.end()

Zwraca **iterator wskazujący za ostatni element**.

To znaczy: nie na ostatni element, ale **tuż za nim** (tzw. *past-the-end iterator*).

👉 it != v.end()

Pętla działa dopóki it **nie osiągnie końca kontenera**.

Gdy iterator zrówna się z end(), kończymy iterację.

Lekcja

Temat: Iteratory. Metoda std::sort, std::find

Iterator to „wskaźnik” na element w kolekcji (np. `vector`, `list`, `map`).

Pozwala przechodzić po elementach oraz je odczytywać lub modyfikować, nie znając wewnętrznej struktury kontenera.

Możesz traktować iterator jak wskaźnik:

- `it = v.begin()` — iterator na pierwszy element
- `it = v.end()` — iterator na element za ostatnim
- `++it` — przejście do kolejnego
- `*it` — odczyt/zmiana wartości

```
#include <iostream>
#include <vector>

int main() {

    std::vector<int> numbers = {10, 20, 30, 40};

    std::cout << "Iteracja po vectorze:" << std::endl;

    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " "; // odczytujemy element
    }

    std::cout << std::endl;
    return 0;
}
```

◆ 1. Metoda `std::sort` – sortowanie

`std::sort` sortuje zakres danych w kontenerach **posiadających dostęp przez indeksy** (np. `vector`, `array`, ale nie `list` i `map`). Nie sortuje map, ponieważ `std::map<int, string>` jest zawsze posortowana po kluczu rosnąco

📌 Wymaga:

```
#include <algorithm>
```

Przykład:

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

int main() {
    // ===== VECTOR =====
    std::vector<int> vec = {5, 1, 9, 3, 7};

    std::sort(vec.begin(), vec.end());

    std::cout << "Vector posortowany rosnaco: ";
    for (int x : vec) std::cout << x << " ";
    std::cout << std::endl;

    // ===== ARRAY =====
    std::array<int, 5> arr = {20, 4, 15, 7, 2};

    std::sort(arr.begin(), arr.end());

    std::cout << "Array posortowany rosnaco: ";
    for (int x : arr) std::cout << x << " ";
    std::cout << std::endl;

    return 0;
}
```

◆ 3. Metoda `std::find` – wyszukiwanie elementu

`std::find` szuka wartości w przedziale **[begin, end)**.

Przykład `std::find` na vector, array, list i string

```
#include <iostream>
#include <vector>
#include <array>
#include <list>
#include <algorithm>
#include <string>

int main() {

    // ===== VECTOR =====
    std::vector<int> vec = {10, 20, 30, 40};
    auto it_vec = std::find(vec.begin(), vec.end(), 30);

    if (it_vec != vec.end())
        std::cout << "Znaleziono w vectorze: " << *it_vec << std::endl;
```

```

// ===== ARRAY =====
std::array<int, 5> arr = {1, 2, 3, 4, 5};
auto it_arr = std::find(arr.begin(), arr.end(), 4);

if (it_arr != arr.end())
    std::cout << "Znaleziono w array: " << *it_arr << std::endl;

// ===== LIST =====
std::list<int> lst = {7, 8, 9, 10};
auto it_list = std::find(lst.begin(), lst.end(), 9);

if (it_list != lst.end())
    std::cout << "Znaleziono w list: " << *it_list << std::endl;

// ===== STRING =====
std::string text = "Hello";
auto it_str = std::find(text.begin(), text.end(), 'e');

if (it_str != text.end())
    std::cout << "Znaleziono w string: " << *it_str << std::endl;
}

```

✓ **std::find** można stosować na:

- vector
- array
- list
- deque
- string
- set (wolne, ale działa)

✗ **std::find** NIE służy do szukania w mapach
(do tego używaj **.find(key)**).

✓ MOŻNA stosować **std::find** na:

Kontener	Można użyć std::find ?	Dlaczego
vector	✓ TAK	Ma iteratory
array	✓ TAK	Ma iteratory
list	✓ TAK	Ma iteratory
deque	✓ TAK	Ma iteratory
string	✓ TAK	Też ma iteratory
set	✓ TAK (ale bez sensu)	Każdy element jest unikalny + wolne O(n)

map

✓ TAK, ALE...

Szuka wartości pary, nie po kluczu!

**1?? 1 osoby do poprawy sprawdzianu
1 osoba nie pisała sprawdzianu**

2 do poprawy kartkówki z funkcji , klas szablonowych

4 osoba nie pisała kartkówki z funkcji , klas szablonowych

na tej lekcji kartkówka z funkcji i klas vector, list, map