PRZEDMIOT: Podstawy programowania

KLASA: 2A gr. 1

Lekcja

Temat: Klasy w C++

Klasa w C++ to podstawowy element programowania obiektowego (OOP), który pozwala na definiowanie własnych typów danych. Jest to szablon lub blueprint, który łączy w sobie dane (pola lub zmienne członkowskie) oraz funkcje (metody), które operują na tych danych. Klasy umożliwiają enkapsulację (ukrywanie szczegółów implementacji), dziedziczenie (ponowne wykorzystanie kodu) i polimorfizm (różne zachowania w zależności od kontekstu).

Przykład 1:

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

class Samochod {
private:
   string marka;
    int rokProdukcji;

public:
    Samochod(std::string m, int r) {
        marka = m;
        rokProdukcji = r;
    }
    void wyswietlInfo() {
       cout << "Marka: " << marka << ", Rok produkcji: " << rokProdukcji << endl;
    }
};

int main() {
    Samochod mojSamochod("Toyota", 2020);
    mojSamochod.wyswietlInfo();
    return 0;
}
```

Kluczowe cechy klas w C++

- **Konstruktory i destruktory**: Konstruktor (np. Samochod()) inicjalizuje obiekt, destruktor (~Samochod()) czyści zasoby po zniszczeniu obiektu.
- **Dziedziczenie**: Możesz tworzyć klasy pochodne, np. class ElektrycznySamochod : public Samochod { ... };.
- **Polimorfizm**: Metody wirtualne (virtual) pozwalają na nadpisywanie zachowań w klasach dziedziczących.
- **Statyczne elementy**: Pola lub metody statyczne (static) należą do klasy, nie do instancji (np. licznik obiektów).

Klasy są podobne do struktur (struct), ale domyślnie w struct elementy są publiczne, a w class prywatne. W praktyce klasy są używane do modelowania rzeczywistych obiektów, co ułatwia pisanie modularnego i utrzymywalnego kodu

Konstruktor

Konstruktor to specjalna metoda klasy, która jest wywoływana automatycznie w momencie tworzenia obiektu (instancji klasy). Jego głównym zadaniem jest inicjalizacja pól klasy, alokacja zasobów (np. pamięci) lub ustawienie początkowych wartości. Konstruktor ma taką samą nazwę jak klasa i nie zwraca żadnej wartości (nawet void nie jest potrzebne).

- **Rodzaje konstruktorów**:
  - **Domyślny**: Bez parametrów, tworzony automatycznie przez kompilator, jeśli nie zdefiniujesz własnego.
  - **Parametryzowany**: Z parametrami, jak w przykładzie poniżej.
  - **Kopiujący**: Kopiuje dane z innego obiektu.
  - **Przenoszący** (w C++11+): Przenosi zasoby z innego obiektu.

Jeśli nie zdefiniujesz konstruktora, kompilator stworzy domyślny.

Destruktor

Destruktor to specjalna metoda klasy, która jest wywoływana automatycznie w momencie niszczenia obiektu (np. gdy obiekt wychodzi poza zakres widoczności lub jest usuwany ręcznie za pomocą delete). Służy do zwalniania zasobów, takich jak pamięć, pliki czy połączenia sieciowe, aby uniknąć wycieków pamięci. Destruktor ma nazwę klasy poprzedzoną tyldą (~) i nie przyjmuje parametrów ani nie zwraca wartości.

- Klasa może mieć tylko jeden destruktor.
- Jeśli nie zdefiniujesz destruktora, kompilator stworzy domyślny, który nic nie robi (chyba że klasa ma pola wymagające czyszczenia).
- Destruktory są szczególnie ważne w klasach zarządzających zasobami (np. wskaźnikami).

Przykład kodu

Przykład 2:

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

class Samochod {
private:
   string marka;
    int rokProdukcji;
    int* numerVIN;  // Przykładowe dynamiczne alokowanie pamięci

public:
    // Konstruktor parametryzowany
    Samochod(string m, int r) {
        marka = m;
        rokProdukcji = r;
        numerVIN = new int;  // Alokacja pamięci
        *numerVIN = 123456;
        std::cout << "Konstruktor wywołany: Obiekt stworzony." <<endl;
    }

    // Destruktor
    ~Samochod() {
        delete numerVIN;  // Zwolnienie pamięci, aby uniknąć wycieku
       cout << "Destruktor wywołany: Obiekt zniszczony." <<endl;
    }
    void wyswietlInfo() {
       cout << "Marka: " << marka << ", Rok produkcji: " << rokProdukcji
                << ", Numer VIN: " << *numerVIN <<endl;
    }
};

int main() {
    {
        Samochod mojSamochod("Toyota", 2020);
        mojSamochod.wyswietlInfo();
}  // Koniec bloku - obiekt niszczony, destruktor wywołany automatycznie

    return 0;
}
```

Wyjaśnienie przykładu:

- **Konstruktor**: Samochod(std::string m, int r) inicjalizuje pola marka i rokProdukcji, alokuje pamięć dla numerVIN i wyświetla komunikat.
- **Destruktor**: ~Samochod() zwalnia pamięć za pomocą delete i wyświetla komunikat. Wywoływany automatycznie na końcu bloku {} w main().

Bez destruktora pamięć po numerVIN nie zostałaby zwolniona, co mogłoby prowadzić do problemów w większych programach.

Przykład 3:

```
#include <iostream>
#include <string>

class Osoba {
private:
    std::string imie;
    std::string nazwisko;
    int wiek;

public:
    Osoba(std::string i, std::string n, int w) {
        imie = i;
        nazwisko = n;
        wiek = w;
    }

    void wyswietlInfo() {
        std::cout << "Imię: " << imie << ", Nazwisko: " << nazwisko << ", Wiek: " << wiek
<< std::endl;
    }

    // Metoda sprawdzająca pełnoletniość (przykład logiki)
    bool jestPelnoletnia() {
        return (wiek >= 18);
    }
};

int main() {
    Osoba mojaOsoba("Jan", "Kowalski", 25);
    mojaOsoba.wyswietlInfo();

    if (mojaOsoba.jestPelnoletnia()) {
        std::cout << "Osoba jest pełnoletnia." << std::endl;
    } else {
        std::cout << "Osoba nie jest pełnoletnia." << std::endl;
    }

    return 0;
}
```

Przykład 4:

```
#include <iostream>
#include <string>

class Dom {
private:
    std::string adres;
    int liczbaPokoi;
    double powierzchnia;  // w metrach kwadratowych

public:
    // Konstruktor parametryzowany
    Dom(std::string a, int p, double pow) {
        adres = a;
        liczbaPokoi = p;
        powierzchnia = pow;
    }

    // Metoda do wyświetlania informacji
    void wyswietlInfo() {
        std::cout << "Adres: " << adres << ", Liczba pokoi: " << liczbaPokoi
                  << ", Powierzchnia: " << powierzchnia << " m²" << std::endl;
    }

    // Metoda sprawdzająca, czy dom jest duży
    bool jestDuzy() {
        return (powierzchnia > 100.0);
    }
};

int main() {
    Dom mojDom("Ul. Główna 123, Warszawa", 4, 120.5);  // Tworzenie obiektu
    mojDom.wyswietlInfo();                              // Wyświetlenie info

    if (mojDom.jestDuzy()) {
        std::cout << "Dom jest duży." << std::endl;
    } else {
        std::cout << "Dom jest mały." << std::endl;
    }
    return 0;
}
```

Lekcja

Temat: Przeciążenie operatorów. Operator unarny i binarny

Operator to np.:

- **+**
- **-**
- *****
- **/**
- **==**
- **<**

Przeciążenie operatorów (ang. operator **overloading**) w C++ to **mechanizm, który pozwala na definiowanie własnego zachowania dla standardowych operatorów (takich jak +, -, *, /, ==, itd.) w kontekście klas lub struktur.** Dzięki temu możesz sprawić, że operatory działają na obiektach twoich własnych typów w sposób podobny do typów wbudowanych, co poprawia czytelność i intuicyjność kodu.

Przykład

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

class Wektor2D {
public:
    int x, y;

    Wektor2D(int  x, int y) : x(x), y(y) {}

    // przeciążenie operatora +
    Wektor2D operator+(const Wektor2D& w) const {
        return Wektor2D(x + w.x, y + w.y);
    }

   Wektor2D add(const Wektor2D& w) const {
        return Wektor2D(x + w.x, y + w.y);
    }
};

int main() {
    Wektor2D a(3, 2);   // pierwszy wektor
    Wektor2D b(5, 1);   // drugi wektor

    Wektor2D c = a + b; // WYWOŁANIE operatora +

    cout << "Wynik dodawania Operator+ (" << c.x << ", " << c.y << ")\n";

    Wektor2D addc = a.add(b);
    cout << "Metoda addc(): (" << addc.x << ", " << addc.y << ")\n";
}
```

Rezultat:

```
Wektor2D a(2, 3);
Wektor2D b(4, 1);

Wektor2D c = a + b; // działa dzięki operator overloading
```

Przeciążanie operatora wypisywania << (cout)

Przykład:

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

class Punkt {
public:
    int x, y;
    Punkt(int x, int y) : x(x), y(y) {}

    // Operator dodawania
    Punkt operator+(const Punkt& p) const{
        return Punkt(x + p.x, y + p.y);
    }

    // Operator odejmowania
    Punkt operator-(const Punkt& p) const{
        return Punkt(x - p.x, y - p.y);
    }

    // Operator porównania ==
    bool operator==(const Punkt& p) const{
        return x == p.x && y == p.y;
    }

};

// operator wypisywania
ostream& operator<<(ostream& out, const Punkt& p) {
    return out << "(" << p.x << ", " << p.y << ")";
}

int main() {
    Punkt a(3, 4);
    Punkt b(1, 2);

    Punkt c = a + b;  // dodawanie
    Punkt d = a - b;  // odejmowanie

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a + b = " << c << endl;
    cout << "a - b = " << d << endl;

    if (a == b)
```

```
        cout << "Punkty są równe\n";
    else
        cout << "Punkty są różne\n";

    return 0;
}
```

Rezultat:

```
Punkt p(3, 5);
cout << p;   // wypisze: (3, 5)
```

🔹 **Operator unarny**

Działa na **jednym** argumencie.

Przykłady (wbudowane w C++):

- -x (negacja liczby)
- ++x (preinkrementacja)
- x-- (postdekrementacja)
- !x (logiczne NIE)

W klasach możesz przeciążyć np.:

```
Punkt operator-() const;    // unarny minus
Punkt operator++();         // ++p
Punkt operator++(int);      // p++
```

Takie operatory mają **jeden** operand.

🔹 **Operator binarny**

Działa na **dwóch** argumentach.

Przykłady:

- a + b
- a - b
- a * b
- a == b
- a < b

W klasie wygląda to tak:

```
Punkt operator+(const Punkt& p) const;
bool operator==(const Punkt& p) const;
```

Operatory **unaryjne (unary)** działają na *jednym* obiekcie:

- - (zmiana znaku)
- ++ (inkrementacja)
- -- (dekrementacja)

Przykład klasy z unarnymi operatorami

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

class  Wektor {
public:
   int x, y;
```

```
    Wektor(int x, int y) : x(x), y(y) {}


    // UNARNY operator - (zmienia znak)
    Wektor operator-() const {
        return Wektor(-x, -y);
    }

    // UNARNY operator ++ (preinkrementacja)
    Wektor& operator++() {
        x++;
        y++;
        return *this;
    }

    // UNARNY operator -- (predekrementacja)
    Wektor& operator--() {
        x--;
        y--;
        return *this;
    }
};

int main() {
    Wektor a(3, 4);

    Wektor b = -a;     // wywołuje operator-()
    cout << "Negacja: (" << b.x << ", " << b.y << ")\n";

    ++a;               // wywołuje operator++()
    cout << "Po ++: (" << a.x << ", " << a.y <<  ")\n";

    --a;               // wywołuje operator--()
    cout << "Po --: (" << a.x << ", " << a.y << ")\n";
}
```

Wynik działania:
Negacja: (-3, -4)
Po ++: (4, 5)
Po --: (3, 4)

Przykład klasy z operatorami binarnymi

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

class Wektor {
public:
   int x, y;
   Wektor(int x, int y) : x(x), y(y) {}


    // BINARNY operator +
    Wektor operator+(const Wektor& w) const{
        return Wektor(x + w.x, y + w.y);
    }
```

```
    // BINARNY operator -
    Wektor operator-(const Wektor& w) const{
        return Wektor(x - w.x, y - w.y);
    }

    // BINARNY operator ==
    bool operator==(const Wektor& w) const{
        return x == w.x && y == w.y;
    }
};

int main() {
    Wektor a(3, 2);
    Wektor b(5, 1);

    Wektor c = a + b;     // wywołuje operator+
    cout << "a + b = (" << c.x << ", " << c.y << ")\n";

    Wektor d = a - b;     // wywołuje operator-
    cout << "a - b = (" << d.x << ", " << d.y << ")\n";

    if (a == b)           // wywołuje operator==
        cout << "a i b są równe\n";
    else
        cout << "a i b NIE są równe\n";
}
```

Lekcja

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

1️⃣ 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"

2️⃣ 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
}
```

3️⃣ 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
}
```

4️⃣ 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.

5️⃣ 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) - vector, list, map

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.

1️⃣ 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 << " ";
}
```

it jest iteratorem, który może wskazywać na element wektora typu vector<int>.
Możesz myśleć o iteratorze jak o wskaźniku:

- *it – daje wartość elementu, na który wskazuje iterator
- ++it – przesuwa iterator do następnego elementu

Przykład:

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

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

    // --- DODAWANIE NA KOŃCU ---
    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;
}
```

2️⃣ 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 pozwala kompilatorowi automatycznie dobrać typ iteratora (std::list<int>::iterator)*/
    auto it = lst.begin();
/* Funkcja std::advance przesuwa iterator o zadaną liczbę kroków (czyli w naszym
przypadku o 1. */
    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 usunieciu 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
```

3️⃣ 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() {
  // Mapa automatycznie sortuje elementy według klucza (rosnąco).
    std::map<int, std::string> mp; // int - typ klucza; std::string - typ wartości


    // --- 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 NAJMNIEJSZEGO KLUCZA (poczatek) ---
    mp.erase(mp.begin());

    std::cout << "\nPo usunieciu najmniejszego klucza:\n";

/*
&p → iterujemy po referencji, czyli nie kopiujemy elementów mapy, tylko odnosimy się
bezpośrednio do ich pamięci. To szybciej i bezpieczniej przy dużych mapach.
W każdej iteracji p wskazuje na jedną parę klucz-wartość w mapie.
*/
    for (auto &p : mp) {
         std::cout <<"Klucz -> "<< p.first << " wartość: " << 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 <<"Klucz -> "<< p.first << " wartość: " << 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 <<"Klucz -> "<< p.first << " wartość: " << p.second << "\n";
    }

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

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

Z tej lekcji będzie kartkówka na następnej lekcji. Jest w kalendarzu zaplanowana na czwartek: 04.12.2025 r.