

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++

- **Konstruktor i destruktor:** 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).
- Destruktry 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;
    }

    // Destraktor
    ~Samochod() {
        delete numerVIN; // Zwolnienie pamięci, aby uniknąć wycieku
        cout << "Destraktor 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 niszczone, destraktor 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.
- **Destraktor:** ~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 jestPełnoletnia() {
        return (wiek >= 18);
    }
};

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

    if (mojaOsoba.jestPełnoletnia()) {
        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** argumentcie.

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.

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) - 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.

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 << " ";  
}
```

`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 usunięciu 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 początku (insert): ";  
    for (int x : v) std::cout << x << " ";  
    std::cout << "\n";  
  
    // --- USUWANIE Z POCZĄTKU ---  
    v.erase(v.begin());
```

```

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

return 0;
}

```

2 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("Ała");
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 usunięciu 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 początku (push_front): ";
for (int x : lst) std::cout << x << " ";
std::cout << "\n";

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

std::cout << "Po usunięciu z początku (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 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 porzucane 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() {
    // 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 (początek) ---
```

```
    mp.erase(mp.begin());
```

```
    std::cout << "\nPo usunięciu 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 usunięciu największego 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 usunięciu 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.

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 <i>wartości pary</i> , nie po kluczu!

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

Lekcja







Temat: Usystematyzowanie materiału

Tabela: Dodawanie i usuwanie w vector, list i map

Legenda:

-  — brak takiej możliwości
-  — jest możliwe
-  — możliwe, ale **nieoptymalne** (kosztowne operacje)

1. std::vector

Operacja	Możliwe?	Metoda / komentarz
Dodaj na początku		insert(v.begin(), x)
Dodaj na końcu		push_back(x)
Dodaj w środku (na pozycji)		insert(iterator, x)
Usuń z początku		erase(v.begin())
Usuń z końca		pop_back()
Usuń ze środka		erase(iterator)

 Vector nie ma *push_front()* ani *pop_front()*.

2. std::list (lista dwukierunkowa)

Operacja	Możliwe?	Metoda
Dodaj na początku		push_front(x)
Dodaj na końcu		push_back(x)
Dodaj w środku		insert(iterator, x)
Usuń z początku		pop_front()
Usuń z końca		pop_back()
Usuń ze środka		erase(iterator)

3. std::map (drzewo RB — uporządkowana mapa)

Operacja	Możliwe?	Metoda / komentarz
Dodaj na początku	✗	brak — mapa jest uporządkowana
Dodaj na końcu	✗	brak — porządek zależy od klucza
Dodaj element	✓	insert({key, val}), m[key] = val
Usuń element o kluczu	✓	erase(key)
Usuń przez iterator	✓	erase(iterator)
Usuń pierwszy element	✓	erase(m.begin())
Usuń ostatni element	✓	auto it = prev(m.end()); erase(it);

📌 W map nie ma pojęcia „początku” i „końca” w sensie kolejki — porządek jest sortowany po kluczu.

Lekcja

Temat: Adresy pamięci i operator &. Wskaźniki i operator * (dereferencja). Dynamiczna alokacja pamięci (new / delete) Referencje: Różnice od wskaźników, referencje jako parametry funkcji, stałe referencje w C++

Każda zmienna w C++ znajduje się **pod jakimś adresem w pamięci RAM**. Operator & pozwala **pobrać adres zmiennej**.

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

```
int main() {

    int x = 10;

    cout << "Wartosc x: " << x << endl;
    cout << "Adres x: " << &x << endl;
    return 0;
}
```

Wskaźnik to **zmienna, która przechowuje adres innej zmiennej**. `int* p`;

Operator `*`

- w deklaracji → mówi, że to wskaźnik
- w użyciu → **odczytuje lub modyfikuje wartość pod adresem**

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

```
int main() {
    int x = 5;
    int* p = &x;
    /*
        x      wartość zmiennej (5)
        &x     adres zmiennej x w pamięci
        p      adres zmiennej x
        *p     wartość znajdująca się pod adresem, który jest w p
    */
```

```
cout << "x = " << x << endl;
cout << "Adres x: " << &x << endl;
cout << "Wskaźnik p: " << p << endl;
cout << "Wartosc pod adresem p (*p): " << *p << endl;
```

`*p = 20;` // zmiana x przez wskaźnik To **NIE** zmienia wskaźnika To zmienia **zmienną x**

```
cout << "Nowa wartosc x: " << x << endl;

return 0;
}
```

Dynamiczna alokacja pamięci (`new` / `delete`) pozwala **tworzyć zmienne w trakcie działania programu**, a nie tylko w czasie kompilacji.

☐ Pojedyncza zmienna

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

int main() {
    int* p = new int; // alokacja
    *p = 42;

    cout << "Wartosc: " << *p << endl;

    delete p; // zwolnienie pamięci
    p = nullptr; // p nie wskazuje już na żadną pamięć (pusty wskaźnik)

    return 0;
}
```

☐ Dynamiczna tablica

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

int main() {
    int n = 5;
    int* tab = new int[n];

    for (int i = 0; i < n; i++) {
        tab[i] = i * 10;
    }

    for (int i = 0; i < n; i++) {
        cout << tab[i] << " ";
    }

    delete[] tab; // UWAGA: delete[]
    tab = nullptr;

    return 0;
}
```

Wskaźnik do funkcji może **przechowywać adres funkcji**, dzięki czemu można:

- przekazywać funkcje jako argumenty
- wybierać funkcję w czasie działania programu

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

int dodaj(int a, int b) {
    return a + b;
}

int odejmij(int a, int b) {
    return a - b;
}

int main() {
    int (*wsk)(int, int); // wskaźnik do funkcji

    wsk = dodaj;
    cout << "Dodawanie: " << wsk(3, 4) << endl; // wywołanie funkcji przez wskaźnik

    wsk = odejmij;
    cout << "Odejmowanie: " << wsk(10, 5) << endl;

    return 0;
}
```

✗ Nigdy nie używaj *p, jeśli p nie wskazuje na nic sensownego

```
int* p;
*p = 10; // BŁĄD – niezdefiniowane zachowanie
```

✓ **Poprawnie:**

```
int x = 10;
int* p = &x;
```

albo

```
int* p = new int;
*p = 10;
```

*p to **dereferencja wskaźnika** – dostęp do wartości znajdującej się pod adresem przechowywanym w p.

```

#include <iostream>
using namespace std;

int main() {
    int x = 10;
    int* p = &x;

    cout << "x = " << x << endl;           // *p == x
    cout << "&x = " << &x << endl;         // p == &x
    cout << "p = " << p << endl;           // p == &x
    cout << "*p = " << *p << endl;         // *p == x
    cout << "&p = " << &p << endl;

    return 0;
}

```

Referencje: Różnice od wskaźników, referencje jako parametry funkcji, stałe referencje.

Referencja to alternatywna nazwa (alias) dla istniejącej zmiennej.

```

int x = 10;
int& ref = x;

```

- **ref to to samo co x**
- nie zajmuje osobnej „logicznej” zmiennej
- każda zmiana **ref** zmienia **x**

```

#include <iostream>
using namespace std;

int main() {
    int x = 10;
    int& r = x;

    r = 20;

    cout << "x = " << x << endl; // 20
    cout << "r = " << r << endl; // 20

    return 0;
}

```

Referencje jako parametry funkcji

✗ Przekazywanie przez wartość

```
void zmien(int a) {  
    a = 100;  
}
```

➡ nie zmienia zmiennej w main

✓ Przekazywanie przez referencję

```
void zmien(int& a) {  
    a = 100;  
}
```

➡ zmienia oryginalną zmienną

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

```
void zwieksz(int& x) {  
    x += 1;  
}
```

```
int main() {  
    int a = 5;  
    zwieksz(a);  
    cout << a << endl; // 6  
    return 0;  
}
```

Stałe referencje (const &) - referencja, przez którą **nie można zmieniać obiektu**

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

```
void wypisz(const int& x) {  
    // x = 10; // błąd kompilacji  
    cout << x << endl;  
}
```

```
int main() {  
    int a = 5;  
    wypisz(a);  
    return 0;  
}
```

Lekcja

Temat: Wyjątki: Try-catch, throw, obsługa błędów, niestandardowe wyjątki

Wyjątki w C++ to mechanizm obsługi błędów, który pozwala na separację normalnego przepływu programu do obsługi sytuacji wyjątkowych (np. błędów w trakcie wykonywania). Zamiast sprawdzania błędów za pomocą kodów zwrotnych (jak w starszych językach), wyjątki umożliwiają "rzucenie" błędu w jednym miejscu i "złapanie" go w innym, co czyni kod bardziej czytelnym i modularnym.

Kluczowe elementy:

- **try**: Blok, w którym umieszczamy kod, który może spowodować błąd.
 - **catch**: Blok(i), które obsługują rzucone wyjątki. Można mieć wiele catch dla różnych typów wyjątków.
 - **throw**: Instrukcja do rzucenia wyjątku.
 - **Obsługa błędów**: Wyjątki są używane do obsługi błędów runtime, np. dzielenie przez zero, brak pamięci, błędy plików.
 - **Niestandardowe wyjątki**: Możesz tworzyć własne klasy dziedziczące po `std::exception` dla bardziej precyzyjnej obsługi.
-
- **try / catch**

Składnia

```
try {  
    // kod, który może spowodować błąd  
}  
catch (typ wyjątku e) {  
    // obsługa błędu  
}
```

Przykład: dzielenie przez zero

```
#include <iostream>  
using namespace std;  
  
int dzielenie(int a, int b) {  
    if (b == 0) {  
        throw "Dzielenie przez zero!"; // throw przerywa wykonywanie funkcji  
    }  
    return a / b;  
}
```



```
int main() {
    try {
        cout << dzielenie(10, 0) << endl;
    }
    catch (const char* msg) { // sterowanie przechodzi do catch
        cout << "Błąd: " << msg << endl; // sterowanie przechodzi do catch
    }
}
```

- **throw**

Instrukcja **throw** w C++ służy do zgłaszania (rzucania) wyjątku, co oznacza **przerwanie normalnego przepływu programu i przekazanie kontroli do mechanizmu obsługi wyjątków (try-catch)**.

Wyjątek może być rzucony w dowolnym miejscu kodu, np. w funkcji, gdy wystąpi błąd (np. niepoprawne dane wejściowe, brak zasobów).

throw może **rzucić wartość dowolnego typu: prymitywnego (int, double), obiektu (np. string), lub niestandardowej klasy wyjątku**. Gdy wyjątek zostanie rzucony, program szuka najbliższego bloku catch, który pasuje do typu rzuconego wyjątku. Jeśli nie znajdzie, program może się zakończyć (wywołując `std::terminate()`).

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

int main() {
    try {
        throw 404;
    }
    catch (int errorCode) {
        cout << "Kod błędu: " << errorCode << endl;
    }
}
```

Wiele bloków catch

Możesz obsługiwać **różne typy błędów inaczej**.

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

int main() {
    try {
        throw 3.14;
    }
    catch (int e) {
        cout << "Błąd typu int" << endl;
    }
}
```

```

    }
    catch (double e) {
        cout << "Błąd typu double: " << e << endl;
    }
}

```

Standardowe wyjątki (std::exception)

Biblioteka: <stdexcept>

Najczęstsze:

- std::runtime_error
- std::logic_error
- std::out_of_range
- std::invalid_argument

Przykład: std::invalid_argument

```

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

int pierwiastek(int x) {
    if (x < 0) {
        throw invalid_argument("Liczba nie może być ujemna");
    }
    return x;
}

int main() {
    try {
        pierwiastek(-5);
    }
    catch (const invalid_argument& e) {
        cout << "Błąd: " << e.what() << endl; // e.what() → opis błędu
    }
}

```

- **Niestandardowe wyjątki:**

Własne (niestandardowe) wyjątki w C++

W C++ możesz tworzyć własne klasy wyjątków, co pozwala na bardziej precyzyjną obsługę błędów. Niestandardowe wyjątki zazwyczaj dziedziczą po klasie bazowej `std::exception` (z nagłówka <exception>), co umożliwia korzystanie z metody `what()` do zwracania opisu błędu. Dzięki temu możesz dodać własne pola (np. kod błędu, dodatkowe informacje) i metody, co ułatwia debugowanie i hierarchiczną obsługę wyjątków.

Kluczowe zasady tworzenia własnych wyjątków:

- **Dziedzicz po `std::exception`** lub jej podklasach (np. `std::runtime_error` dla błędów runtime).
- **Nadpisz metodę `const char* what() const noexcept override`**; – zwraca opis błędu.
- **Konstruktor powinien inicjalizować pola i komunikat.**
- Wyjątki powinny być lekkie (unikaj złożonych obiektów), bo są kopiowane podczas propagacji.
- **Używaj ich w bloku `throw`, a łap w `catch` przez referencję (`const MyException& e`).**
- **Hierarchia**: Możesz tworzyć drzewo wyjątków, np. bazowy `AppError`, a pod nim `FileError`, `NetworkError` – łap je w kolejności od najbardziej specyficznych do ogólnych.

Zalety: Lepsza czytelność kodu, możliwość dodania kontekstu (np. wartość powodująca błąd), łatwiejsza obsługa w dużych aplikacjach.

Wady: Dodatkowy kod do napisania; nie używaj do normalnego sterowania przepływem (tylko do błędów).

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

```
class MyException : public exception {
public:
    const char* what() const noexcept override {
        return "To jest mój własny wyjątek!";
    }
};

void test() {
    throw MyException();
}

int main() {
    try {
        test();
    }
    catch (const MyException& e) {
        cout << e.what() << endl;
    }
}
```

Obsługa błędów vs wyjątki

Bez wyjątków

```
bool dzielenie(int a, int b, int& wynik) {
```

```
if (b == 0) return false;
wynik = a / b;
return true;
}
```

Z wyjątkami (czytelniej)

```
int dzielenie(int a, int b) {
    if (b == 0) throw runtime_error("Dzielenie przez zero");
    return a / b;
}
```

Wyjątki są lepsze gdy:

- błąd jest rzadki
- nie chcesz sprawdzać if po każdym wywołaniu
- błąd jest krytyczny logicznie

Lekcja

Temat: Wielowątkowość w C++: Podstawy concurrency

Wielowątkowość (multithreading) w C++ pozwala na wykonywanie wielu zadań jednocześnie w ramach jednego procesu, co poprawia wydajność aplikacji, zwłaszcza na wielordzeniowych procesorach.

Biblioteka <thread> (od C++11) dostarcza narzędzi do tworzenia i zarządzania wątkami.

Concurrency odnosi się do równoległego wykonywania zadań, ale wymaga synchronizacji, aby uniknąć problemów jak race conditions (konflikty dostępu do współdzielonych zasobów).

Kluczowe pojęcia:

- **std::thread**: Klasa do tworzenia i zarządzania wątkami. Zastosowanie: Uruchamianie funkcji w tle, np. obliczenia równoległe, obsługa I/O bez blokowania głównego wątku.
- **Mutexy** (std::mutex): Mechanizm blokady wzajemnej wykluczania. Zastosowanie: Ochrona współdzielonych danych przed jednoczesnym dostępem (tylko jeden wątek na raz).
- **Condition variables** (std::condition_variable): Służą do synchronizacji wątków poprzez sygnalizowanie zdarzeń. Zastosowanie: Oczekiwanie na spełnienie warunku (np. dane gotowe), w połączeniu z mutexem, aby uniknąć busy-waiting (pętli sprawdzającej).

std::thread: Tworzenie wątków

Zastosowanie:

Rozdzielanie zadań, np. w aplikacjach serwerowych (obsługa wielu klientów), symulacjach czy przetwarzaniu danych.

Przykład: Dwa wątki liczące niezależnie.

```
#include <iostream>
#include <thread>

void funkcjaWatku(int id) {
    for (int i = 0; i < 5; ++i) {
        std::cout << "Wątek " << id << " liczy: " << i << std::endl;
    }
}

int main() {
    std::thread watek1(funkcjaWatku, 1);
    std::thread watek2(funkcjaWatku, 2);

    watek1.join();
    watek2.join();

    std::cout << "Główny wątek zakończony." << std::endl;
    return 0;
}
```

Wyjaśnienie pomieszanego wyniku w przykładzie z dwoma wątkami

Twój output z wykonania kodu (przykład z dwoma wątkami liczącymi niezależnie) jest pomieszany, ponieważ w C++ operacje na strumieniu `std::cout` (lub innych strumieniach wyjścia) nie są **thread-safe bez dodatkowej synchronizacji**. Oznacza to, że gdy dwa (lub więcej) wątki próbują jednocześnie pisać do `cout`, ich dane mogą się przeplatać w nieprzewidywalny sposób. To klasyczny przykład **race condition (wyścig wątków)** w kontekście **współdzielonego zasobu** (tutaj: strumień wyjścia).

Dlaczego tak się dzieje?

- **Brak atomowości:** Instrukcja `std::cout << "Wątek " << id << " liczy: " << i << std::endl;` **nie jest wykonywana jako pojedyncza, niepodzielna operacja**. Pod spodem jest to sekwencja kroków (np. zapisywanie "Wątek ", potem `id`, itd.), które mogą być przerwane przez scheduler systemu operacyjnego. Wątek 1 może zacząć pisać, potem system przełączy na wątek 2, który wtrąci swoje dane, i tak dalej.
- **Równoległe wykonanie:** Wątki (`std::thread`) działają naprawdę równoległe (na wielordzeniowym CPU) lub **quasi-równoległe** (przełączanie kontekstu). Kolejność wykonania zależy od systemu, obciążenia, a nawet losowości – stąd output może być inny za każdym uruchomieniem.
- **Brak synchronizacji:** W kodzie nie ma **mutexa ani innego mechanizmu blokującego dostęp do `cout`**. Dlatego wątki **"walczą" o dostęp**, co prowadzi do pomieszania (np. "Wątek Wątek 1 liczy: 0" – to fragmenty z obu wątków sklejone razem).

Aby uniknąć pomieszania, użyj `std::mutex` do ochrony `cout`. Mutex zapewni, że tylko jeden wątek na raz pisze do strumienia.

Zmodyfikowany przykład kodu:

```
#include <iostream>
```

```
#include <thread>
#include <mutex>
```

```
std::mutex coutMutex; // Mutex do ochrony cout
```

```
void funkcjaWatku(int id) {
    for (int i = 0; i < 5; ++i) {
        std::lock_guard<std::mutex> guard(coutMutex); // Automatyczna blokada (RAII)
        std::cout << "Wątek " << id << " liczy: " << i << std::endl;
    }
}
```

```
int main() {
    std::thread watek1(funkcjaWatku, 1);
    std::thread watek2(funkcjaWatku, 2);

    watek1.join();
    watek2.join();

    std::cout << "Główny wątek zakończony." << std::endl;
    return 0;
}
```

- Wyjaśnienie modyfikacji: `std::lock_guard` blokuje mutex na czas pisania całej linii, co czyni operację atomową. Output będzie teraz uporządkowany (np. wszystkie linie wątku 1, potem 2, lub odwrotnie – ale bez mieszania wewnątrz linii).
- Zaleta: Unika race condition, ale kosztem wydajności (wątki czekają na siebie nawzajem).
- Alternatywy: Użyj `std::osyncstream` (C++20) dla thread-safe cout, lub buforuj output w stringu per wątek i wypisz na końcu.

`std::mutex coutMutex;` służy do synchronizacji dostępu do `std::cout` między wątkami, czyli chroni wypisywanie na konsolę przed jednoczesnym użyciem przez kilka wątków.

Problem bez mutexa ❌

`std::cout` **nie jest bezpieczny wątkowo**.

Gdyby **dwa wątki pisały jednocześnie**, mogłoby dojść do:

- przeplatania się tekstu
- nieczytelnego wyniku, np.:

Wątek 1 liczy: Wątek 2 liczy: 00

Wątek 1 liczy: 1

Wątek 2 liczy: 1

Co robi `std::mutex coutMutex;` ✅



`std::mutex coutMutex;`

- tworzy **muteks (blokadę)**,
- który pozwala **tylko jednemu wątkowi naraz** wejść do sekcji krytycznej (tu: wypisywanie na cout).

Jak działa `std::lock_guard` 🔒

`std::lock_guard<std::mutex> guard(coutMutex);`

To jest **RAII** (Resource Acquisition Is Initialization):

-  **blokuje mutex** w konstruktorze
-  **odblokuje mutex** automatycznie w destruktorze (gdy wyjdzie poza zakres {})

Mutexy: Synchronizacja dostępu

Zastosowanie:

Zapobieganie race conditions, np. w licznikach współdzielonych, bazach danych w pamięci.

Przykład: Współdzielony licznik chroniony mutexem.

```
#include <iostream>
#include <thread>
#include <mutex>

int licznik = 0; // Współdzielona zmienna
std::mutex mtx; // Mutex do ochrony

void inkrementuj(int id) {
    for (int i = 0; i < 1000; ++i) {
        mtx.lock(); // Blokuje dostęp
        ++licznik;
        mtx.unlock(); // Odblokuje
    }
    std::cout << "Wątek " << id << " zakończony." << std::endl;
}

int main() {
    std::thread watek1(inkrementuj, 1);
    std::thread watek2(inkrementuj, 2);

    watek1.join();
    watek2.join();

    std::cout << "Ostateczna wartość licznika: " << licznik << std::endl; // Powinno być 2000
    return 0;
}
```

Wyjaśnienie: Bez mutexu licznik mógłby być błędny (race condition). Mutex zapewnia wyłączny dostęp. Lepiej używać `std::lock_guard` dla RAII (automatyczne unlock):

Condition variables: Sygnalizowanie zdarzeń

Zastosowanie:

Koordinacja wątków, np. producent-konsument (jeden produkuje dane, drugi czeka), symulacje z zależnościami.

Przykład: Producent sygnalizuje konsumentowi, że dane są gotowe.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
```

```

std::mutex mtx;
std::condition_variable cv;
bool daneGotowe = false; // Warunek

void producent() {
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Symulacja pracy
    {
        std::lock_guard<std::mutex> lock(mtx);
        daneGotowe = true;
    }
    cv.notify_one(); // Sygnalizuje jednemu czekającemu wątkowi
    std::cout << "Producent: Dane gotowe." << std::endl;
}

void konsument() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return daneGotowe; }); // Czeka na warunek
    std::cout << "Konsument: Otrzymano dane." << std::endl;
}

int main() {
    std::thread prod(producent);
    std::thread kons(konsument);

    prod.join();
    kons.join();

    return 0;
}

```

Wyjaśnienie: Konsument czeka (wait()) aż producent ustawi daneGotowe i powiadomi (notify_one()). Używa unique_lock dla elastyczności (można unlock/lock wielokrotnie). To unika pętli sprawdzającej, oszczędzając CPU.

Podstawy concurrency: Wskazówki i problemy

- **Zalety:** Szybsze wykonanie (równoległość), responsywność aplikacji.
- **Problemy:** Race conditions, deadlocks (wzajemne blokady mutexów), livelocks. Rozwiązania: Używać atomików (std::atomic), future/promise dla asynchroniczności.
- **Biblioteki:** <thread>, <mutex>, <condition_variable>, <atomic>, <future>.
- **Uwagi:** Zawsze używaj join() lub detach() dla wątków. Testuj na wielordzeniowych systemach. Dla zaawansowanego: Użyj std::async dla prostszej asynchroniczności.

Lekcja

Temat: Zarządzanie pamięcią RAII w C++

RAII (Resource Acquisition Is Initialization) to kluczowa technika w C++, która zapewnia automatyczne zarządzanie zasobami takimi jak

- pamięć,
- pliki
- czy połączenia sieciowe.

Zasada polega na tym, że **zasób jest nabywany (alokowany) podczas inicjalizacji obiektu (w konstruktorze)**, a **zwalniany automatycznie w destruktorze obiektu**. Dzięki temu unikamy wycieków pamięci – nie trzeba ręcznie wywoływać `delete` czy `close`, bo **destruktor robi to za nas**, gdy obiekt wychodzi poza zakres widoczności (np. koniec bloku kodu).

RAII jest podstawą inteligentnych wskaźników (smart pointers), które **automatyzują zarządzanie pamięcią dynamiczną**.

W C++ standardowa biblioteka `<memory>` dostarcza trzy główne typy:

- `unique_ptr`,
- `shared_ptr`,
- `weak_ptr`

Zapobiegają wyciekom pamięci, błędom jak podwójne usuwanie czy wiszące wskaźniki.

1. `unique_ptr` - Wyłączna własność

Opis: Reprezentuje wyłączone posiadanie obiektu. Tylko jeden `unique_ptr` może wskazywać na dany obiekt w danym momencie. Nie można go kopiować (bo to by oznaczało współdzielenie), ale można przenosić własność (używając `std::move`). Destruktor automatycznie wywołuje `delete` na obiekcie.

Zalety: Lekki, efektywny, gwarantuje unikalność.

Kiedy używać: Gdy obiekt ma jednego właściciela, np. w lokalnych zmiennych.

Przykład

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

```
int main() {
/*
```

Tworzy miejsce w pamięci na obiekt typu `int` i inicjalizuje. Odpowiednik `new int(42)`

```

*/
    unique_ptr<int> ptr = make_unique<int>(42);
/*
    Dostęp jak do wskaźnika.
    ptr to wskaźnik (adres), a *ptr to wartość znajdująca się pod tym adresem
*/
    cout << "Wartość: " << *ptr << endl;

/*
    Przenoszenie własności, a przeniesienie oznacza oddanie własności i wyczyszczenie starego
    wskaźnika.
    std::move(ptr)- nie przenosi danych, rzuca ptr na r-value, mówi kompilatorowi:
    „Ten obiekt można określić z zasobu”
*/
    unique_ptr<int> ptr2 = move(ptr); // ptr staje się nullptr

    if (ptr == nullptr) { // jeśli wskaźnik, który na nic nie wskazuje
        cout << "ptr jest teraz pusty (nie wskazuje na obiekt) " << endl;
    }
    cout << "Wartość w ptr2: " << *ptr2 << endl;

    // Na końcu main, destruktor unique_ptr automatycznie usuwa obiekt
    return 0;
}

```

Wynik:

Wartość: 42

ptr jest teraz pusty

Wartość w ptr2: 42

2. **shared_ptr** - Wspólna własność

Opis: Pozwala na współdzielone posiadanie obiektu. Używa licznika referencji (reference counting) – każdy **shared_ptr** zwiększa licznik, a gdy spada do zera, obiekt jest usuwany (delete). Można kopiować wskaźnik.

Zalety: Bezpieczne współdzielenie, automatyczne zarządzanie cyklem życia.

Kiedy używać: Gdy obiekt ma wielu właścicieli, np. w strukturach danych jak grafy.

Przykład

```

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

```

```

int main() {
    /*

```

Tworzy obiekt int o wartości 100 oraz obiekt sterujący (licznik referencji) w jednym kawałku pamięci

shared_ptr to inteligentny wskaźnik, który pozwala wielu wskaźnikom współdzielić ten sam obiekt

shared_ptr „trzyma” **ten sam licznik**. Licznik mówi: *ilu właścicieli aktualnie istnieje*

```

*/
shared_ptr<int> ptr1 = make_shared<int>(100);
cout << "Wartość w ptr1: " << *ptr1 << endl;
cout << "Liczba referencji: " << ptr1.use_count() << endl; // 1

// Kopiowanie – współdzielenie
shared_ptr<int> ptr2 = ptr1;
cout << "Wartość w ptr2: " << *ptr2 << endl;
cout << "Liczba referencji: " << ptr1.use_count() << endl; // 2

// Modyfikacja wpływa na oba
*ptr2 = 200;
cout << "Wartość w ptr1 po zmianie: " << *ptr1 << endl; // 200

// Na końcu, gdy oba ptr wyjdą z zakresu, licznik spadnie do 0 i obiekt zostanie usunięty
return 0;
}

```

Wynik:

Wartość w ptr1: 100
 Liczba referencji: 1
 Wartość w ptr2: 100
 Liczba referencji: 2
 Wartość w ptr1 po zmianie: 200

3. **weak_ptr** - Słaba referencja (bez własności)

Opis: To "słabe" odniesienie do obiektu zarządzanego przez `shared_ptr`. Nie zwiększa licznika referencji, więc nie przedłuża życia obiektu. Służy do sprawdzania, czy obiekt jeszcze istnieje (używając `lock()` do uzyskania `shared_ptr`). Zapobiega cyklom referencji (gdy dwa obiekty wskazują na siebie nawzajem i nie mogą być usunięte).

Zalety: Unika cykli, pozwala obserwować bez posiadania.

Kiedy używać: W cache'ach, obserwatorach lub do łamania cykli w grafach.

std::weak_ptr to:

- „obserwator” obiektu,
- bez prawa własności,
- bez wpływu na czas życia obiektu.

On **tylko patrzy**, czy obiekt jeszcze istnieje. **NIGDY nie usuwa obiektu** i **NIGDY go nie utrzymuje przy życiu**.

Przykład

```

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

```

```

struct Node {
    shared_ptr<Node> next; // referencja do następnego węzła. To mogłoby spowodować
    cykl
    weak_ptr<Node> weak_next; // referencja – NIE jest właścicielem obiektu. Użyj weak,
    by uniknąć cyklu
    ~Node() { cout << "Usuwanie Node" << endl; }
};

int main() {
    shared_ptr<int> shared = make_shared<int>(50);
    cout << "Liczba referencji: " << shared.use_count() << endl; // 1

    weak_ptr<int> weak = shared;
    cout << "Liczba referencji (nadal): " << shared.use_count() << endl; // 1, bo weak nie
    zwiększa

    // Sprawdzanie, czy obiekt istnieje
    if (auto locked = weak.lock()) { // lock() zwraca shared_ptr jeśli istnieje
        cout << "Wartość przez weak: " << *locked << endl;
        cout << "Liczba referencji podczas lock: " << shared.use_count() << endl; // 2
    } else {
        cout << "Obiekt już nie istnieje" << endl;
    }

    // Reset shared – obiekt zostanie usunięty
    shared.reset();
    if (weak.expired()) {
        cout << "weak jest expired (obiekt usunięty)" << endl;
    }

    return 0;
}

```

Wynik:

Liczba referencji: 1
 Liczba referencji (nadal): 1
 Wartość przez weak: 50
 Liczba referencji podczas lock: 2
 weak jest expired (obiekt usunięty)