

PRZEDMIOT: Informatyka

Lekcja 1

Temat: Język niskiego poziomu i wysokiego poziomu.
Operacje wejścia/wyjścia, C++

♦ Język niskiego poziomu (Low-level language)

Definicja:

- **Jest bliski językowi maszynowemu, czyli instrukcjom bezpośrednio wykonywanym przez procesor.**
- Programista musi znać szczegóły działania sprzętu, takie jak rejestry, adresy pamięci czy operacje bitowe.

Przykłady:

- **Kod maszynowy** (ciąg zer i jedynek)
- **Assembler / język assemblera**

Cechy:

- Trudny do pisania i czytania dla człowieka
- Bardzo szybki w wykonaniu
- Daje pełną kontrolę nad sprzętem

Prosty przykład w NASM, Netwide Assembler (Linux) – wypisanie znaku **A**

```
section .data

    znak db 'A'      ; jeden znak do wyświetlenia


section .text

    global _start


_start:

    mov edx, 1        ; długość danych = 1 znak
    mov ecx, znak      ; adres znaku
    mov ebx, 1         ; stdout
    mov eax, 4         ; syscall: write
    int 0x80           ; wywołanie systemowe


    mov eax, 1         ; syscall: exit
    int 0x80
```

♦ Język wysokiego poziomu (High-level language)

Definicja:

- **Jest zbliżony do języka naturalnego i abstrakcyjny względem sprzętu.**
- Programista nie musi znać szczegółów działania procesora czy pamięci.

Przykłady:

- C, C++, Java, Python, JavaScript, PHP

Cechy:

- Łatwy do nauki i czytania
- Program jest przenośny między różnymi komputerami
- Wydajność może być niższa niż w językach niskiego poziomu (ale kompilatory/interpretery bardzo to optymalizują)

♦ Dlaczego C++ jest językiem wysokiego poziomu:

- Składnia jest czytelna i zbliżona do języka naturalnego (`if`, `for`, `while`, `class` itp.).
- Programista nie musi znać szczegółów działania procesora, by tworzyć aplikacje.
- Programy są przenośne między różnymi systemami.

♦ Dlaczego ma cechy niskiego poziomu:

- Pozwala na **bezpośrednią manipulację pamięcią** przez wskaźniki.
- Możesz używać **instrukcji niskiego poziomu**, np. operacje bitowe.
- Nadaje się do tworzenia sterowników, systemów operacyjnych, gier wymagających wydajności.

♦ Operacje wejścia/wyjścia w C++

- **Operacje wejścia/wyjścia (I/O)** pozwalają programowi **odczytywać dane od użytkownika** (wejście) lub **wyświetlać dane na ekranie** (wyjście).
- W C++ realizuje się je głównie za pomocą **strumieni** z biblioteki `<iostream>`.

♦ Co zawiera `<iostream>`

1. Strumienie wejścia/wyjścia:

- `std::cin` – standardowe wejście (klawiatura)
- `std::cout` – standardowe wyjście (ekran)
- `std::cerr` – strumień błędów (niebuforowany, na ekran)
- `std::clog` – strumień logów (buforowany, na ekran)

2. Funkcje i operatory związane ze strumieniami:

- `<<` – operator wyjścia
- `>>` – operator wejścia

3. Typy strumieniowe:

- `std::ostream` – bazowy typ dla wyjścia
- `std::istream` – bazowy typ dla wejścia

4. Manipulatory strumieniowe:

- `std::endl` – nowa linia + opróżnienie bufora
- `std::flush` – opróżnienie bufora strumienia
- `std::setw()`, `std::setprecision()` – formatowanie wyjścia (po dołączeniu `<iomanip>`)

♦ `using namespace std;`

- W C++ `std` to **standardowy namespace**, czyli przestrzeń nazw dla biblioteki standardowej C++.
- Zawiera wszystko, co pochodzi z `<iostream>`, `<vector>`, `<string>` itd.
- Dzięki temu nie musisz pisać za każdym razem
 - `std::cout`,
 - `std::string`,
 - `std::vector`.

♦ Dlaczego **main()**

1. Punkt wejścia programu

- Kiedy uruchamiasz program, system operacyjny szuka funkcji **main()** i zaczyna wykonywać kod właśnie stamtąd.

2. Zwracanie wartości typu **int**

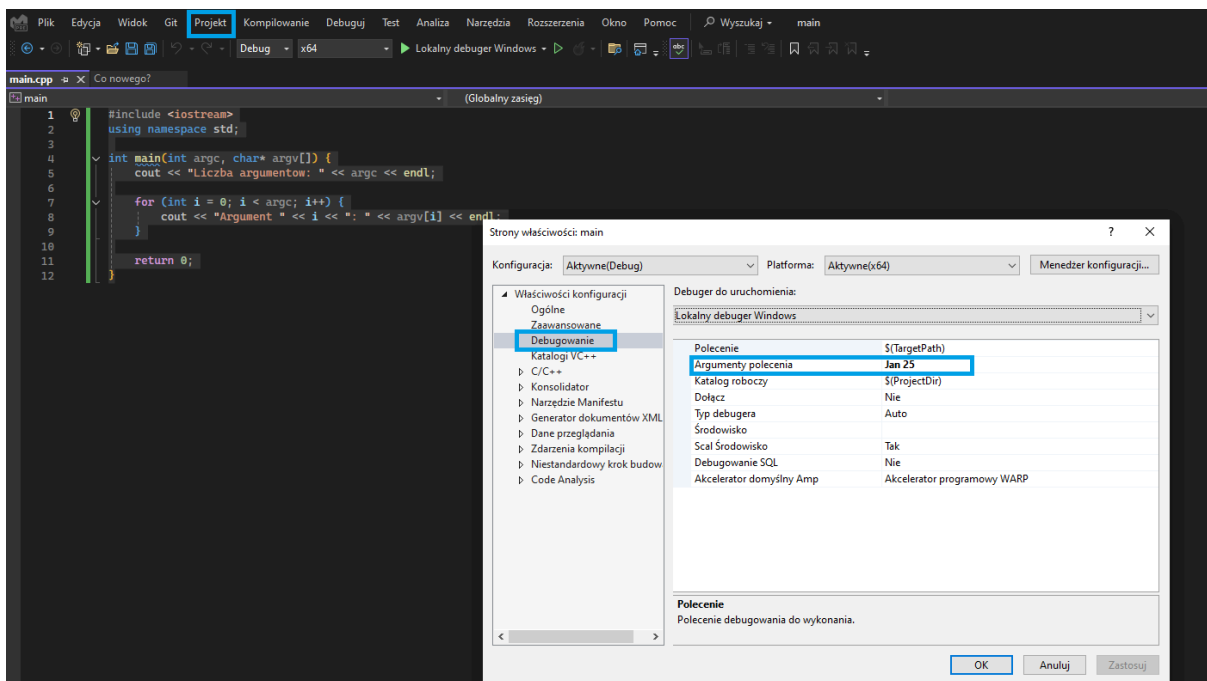
- **int main()** oznacza, że funkcja zwraca liczbę całkowitą.
- System operacyjny interpretuje tę wartość jako **kod zakończenia programu**:
 - **0** → program zakończył się sukcesem
 - inna liczba → program zakończył się błędem

3. Alternatywne formy **main()**

int main(int argc, char* argv[]) – przyjmuje argumenty z linii poleceń.

Kompilacja programu z funkcją main z argumentami wykonuje się dodanie argumentów u ustawieniach właściwości projektu:

Projekt/Właściwości/Debugowanie w opcji **Argumenty polecenia** należy wpisać przykładowe dane np.: Jan 25



♦ 1. Typy podstawowe (proste)

| Typ | Opis | Przykład wartości |
|------------------------|--|-------------------|
| <code>int</code> | Liczby całkowite | 0, 10, -5 |
| <code>short</code> | Krótsze liczby całkowite | 0, 100 |
| <code>long</code> | Dłuższe liczby całkowite | 1000, -5000 |
| <code>long long</code> | Bardzo duże liczby całkowite | 1000000000 |
| <code>unsigned</code> | Liczby całkowite dodatnie tylko | 0, 100 |
| <code>float</code> | Liczby zmiennoprzecinkowe (pojedyncza precyzja, około 7 cyfr znaczących) | 3.14, -0.5 |
| <code>double</code> | Liczby zmiennoprzecinkowe (podwójna precyzja, około 15 cyfr znaczących) | 3.14159 |
| <code>char</code> | Pojedynczy znak | 'a', 'Z', '5' |
| <code>bool</code> | Wartość logiczna | true, false |

♦ 2. Typy złożone

| Typ | Opis | Przykład |
|--|------------------------------------|------------------------------|
| <code>array</code> | Tablica elementów tego samego typu | <code>int tab[5];</code> |
| <code>string</code> (z <code><string></code>) | Ciąg znaków | <code>"Hello"</code> |

♦ 3. Typy wskaźnikowe i referencje

| Typ | Opis | Przykład |
|-----------------------|--------------------------------|---------------------------------|
| <code>int*</code> | Wskaźnik na int | <code>int* ptr = &x;</code> |
| <code>double*</code> | Wskaźnik na double | <code>double* dp;</code> |
| <code>int&</code> | Referencja (alias) do zmiennej | <code>int& ref = x;</code> |

♦ 4. Typy specjalne

| Typ | Opis |
|----------------------|---|
| <code>void</code> | Brak wartości (funkcja nic nie zwraca) |
| <code>auto</code> | Automatyczne określenie typu przez kompilator |
| <code>nullptr</code> | Stała wskaźnikowa oznaczająca „brak adresu” |

♦ Różnice między `struct` a `class` w C++

1. Domyślny dostęp do pól i metod

- w `struct` → domyślnie **public**
- w `class` → domyślnie **private**

2. Zastosowanie historyczne

- `struct` – kiedyś używane głównie jako prosty „koszyk” danych (np. rekord z polami),
- `class` – do programowania obiektowego (metody, enkapsulacja, dziedziczenie).
→ Ale w nowoczesnym C++ oba są prawie tym samym – różnica to głównie **domyślny poziom dostępu**.

3. Dziedziczenie

- w `struct` → domyślnie **publiczne**
- w `class` → domyślnie **prywatne**


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

```
struct Punkt {
    int x;
    int y;
};
```

```
class Prostokat {
    int szerokosc;
    int wysokosc;
```

```
public:
```

```
    Prostokat(int s, int w) {
        szerokosc = s;
        wysokosc = w;
    }
```

```
    int pole() {
        return szerokosc * wysokosc;
    }
};
```

```
int main() {
    Punkt p1;
    p1.x = 10;
    p1.y = 20;
```

```
    cout << "Punkt: (" << p1.x << ", " << p1.y << ")" << endl;
```

```
    Prostokat pr(5, 3);
```

```
    cout << "Pole prostokata: " << pr.pole() << endl;
```

```
}
```

Lekcja 2

Temat: Instrukcje warunkowe. Pętle: For, while, do-while; break, continue; pętle zagnieżdżone.

♦ Instrukcje warunkowe

1. if

Podstawowa instrukcja warunkowa:

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

int main() {
    int x = 10;

    if (x > 5) {
        cout << "x jest większe od 5" << endl;
    }

    return 0;
}
```

2. if...else

Dodanie alternatywnej ścieżki, jeśli warunek nie jest spełniony:

```
int x = 3;

if (x > 5) {
    cout << "x jest większe od 5" << endl;
} else {
    cout << "x jest mniejsze lub równe 5" << endl;
}
```

3. if...else if...else

Sprawdzenie wielu warunków:

```
int x = 0;
```

```

if (x > 0) {
    cout << "Liczba dodatnia" << endl;
} else if (x < 0) {
    cout << "Liczba ujemna" << endl;
} else {
    cout << "Liczba równa zero" << endl;
}

```

4. switch

Instrukcja warunkowa do wyboru jednej z wielu opcji (gdy sprawdzamy wartość jednej zmiennej):

```

int dzien = 3;

switch (dzien) {
    case 1:
        cout << "Poniedziałek" << endl;
        break;
    case 2:
        cout << "Wtorek" << endl;
        break;
    case 3:
        cout << "Środa" << endl;
        break;
    default:
        cout << "Nieznany dzień" << endl;
}

```

break; zatrzymuje wykonanie dalszych przypadków – bez niego program przechodziłby dalej

5. Operator warunkowy (ternary operator)

Skrócona forma if...else:

```

int x = 7;
string wynik = (x % 2 == 0) ? "Parzysta" : "Nieparzysta";

cout << wynik << endl;

```

6. if z inicjalizacją, co pozwala zdefiniować zmienną w zakresie warunku:

```

if (int x = funkcja(); x > 0) {
    // kod, jeśli x > 0
}

```

```
}
```

1. Inkrementacja

To **zwiększenie wartości zmiennej o 1**.

- **preinkrementacja** `++x` – najpierw zwiększa, potem używa wartości,
- **postinkrementacja** `x++` – najpierw używa wartości, potem zwiększa.

2. Dekrementacja

To **zmniejszenie wartości zmiennej o 1**

- **predekrementacja** `--x`,
- **postdekrementacja** `x--`.

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

```
int main() {
    int a = 5;
```

```
    cout << "Preinkrementacja: " << ++a << endl; // najpierw +1 → 6
```

```
    cout << "Postinkrementacja: " << a++ << endl; // używa 6, potem +1 →
```

```
wyświetli 6, ale a = 7
```

```
    cout << "Wartość po: " << a << endl; // 7
```

```
    int b = 5;
```

```
    cout << "Predekrementacja: " << --b << endl; // najpierw -1 → 4
```

```
    cout << "Postdekrementacja: " << b-- << endl; // używa 4, potem -1 →
```

```
wyświetli 4, ale b = 3
```

```
    cout << "Wartość po: " << b << endl; // 3
```

```
    return 0;
```

```
}
```

1. Rodzaje pętli

Pętla for

- **Opis:** Pętla for jest używana, gdy znamy liczbę iteracji z góry. Składa się z trzech części: inicjalizacji, warunku i aktualizacji.

Składnia:

```
for (inicjalizacja; warunek; aktualizacja) {  
    // kod do wykonania  
}
```

Przykład:

```
#include <iostream>  
using namespace std;  
int main() {  
    for (int i = 1; i <= 5; i++) {  
        cout << i << " ";  
    }  
    return 0; // Wypisze: 1 2 3 4 5  
}
```

- **Zastosowanie:** Iterowanie po sekwencji (np. tablicach, liczenie).

Pętla while

- **Opis:** Pętla while wykonuje kod, dopóki warunek jest prawdziwy. Warunek sprawdzany jest przed każdą iteracją.

Składnia:

```
while (warunek) {  
    // kod do wykonania  
}
```

Przykład:

```
#include <iostream>  
using namespace std;  
int main() {  
    int i = 1;  
    while (i <= 5) {  
        cout << i << " ";  
        i++;  
    }  
}
```

```
return 0; // Wypisze: 1 2 3 4 5
}
```

- **Zastosowanie:** Gdy liczba iteracji nie jest znana z góry (np. wczytywanie danych do momentu wprowadzenia określonej wartości).

Pętla do-while

- **Opis:** Podobna do while, ale warunek sprawdzany jest po wykonaniu kodu, co gwarantuje przynajmniej jedno wykonanie pętli.

Składnia:

```
do {
    // kod do wykonania
} while (warunek);
```

Przykład:

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do {
        cout << i << " ";
        i++;
    } while (i <= 5);
    return 0; // Wypisze: 1 2 3 4 5
}
```

- **Zastosowanie:** Gdy chcemy zapewnić wykonanie kodu przynajmniej raz (np. menu użytkownika).

Instrukcje break i continue

- **break:** Natychmiast przerywa pętlę i przechodzi do kodu po pętli.

Przykład:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    cout << i << " "; // Wypisze: 1 2 3 4
}
```

- **continue:** Pomija resztę kodu w bieżącej iteracji i przechodzi do następnej.

Przykład:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " "; // Wypisze: 1 2 4 5
}
```

Pętle zagnieżdżone

- **Opis:** Pętla wewnątrz innej pętli. Używana do pracy z danymi wielowymiarowymi (np. tablice 2D) lub generowania wzorców.

Przykład (trójkąt z gwiazdek):

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= i; j++) {
            cout << "* ";
        }
        cout << endl;
    }
    return 0;
    /* Wypisze:
    *
    * *
    * * *
    * * * *
    * * * * *
    */
}
```

- **Zastosowanie:** Przetwarzanie macierzy, generowanie wzorców, iterowanie po złożonych strukturach danych.

Znaczenie zakresu zmiennych lokalnych i globalnych przy pętlach

- **Zmienne lokalne:**
 - Deklarowane wewnątrz funkcji lub bloku kodu (np. w pętli for).
 - Są widoczne tylko w bloku, w którym zostały zadeklarowane.

Przykład w pętli:

```
for (int i = 0; i < 5; i++) { // i jest lokalne dla pętli
    cout << i << " ";
}
// cout << i; // Błąd! i nie jest dostępne poza pętlą
```

- **Wpływ na pętle:** Zmienne lokalne w pętlach (np. licznik i) są niszczone po zakończeniu pętli, co zapobiega konfliktom w innych częściach programu. W pętlach zagnieżdżonych każda pętla może mieć własne zmienne lokalne o tej samej nazwie bez kolizji.

- **Zmienne globalne:**

- Deklarowane poza wszystkimi funkcjami, dostępne w całym programie.

Przykład:

```
#include <iostream>
using namespace std;
int counter = 0; // Zmienna globalna
int main() {
    for (int i = 0; i < 5; i++) {
        counter++;
    }
    cout << counter; // Wypisze: 5
    return 0;
}
```

- **Wpływ na pętle:** Zmienne globalne mogą być używane w pętlach, ale należy ich unikać, ponieważ:
 - Mogą prowadzić do niezamierzonych zmian w innych częściach programu.
 - Utrudniają debugowanie (np. trudniej znaleźć, gdzie zmienna została zmieniona).
 - Są mniej bezpieczne, bo każda funkcja/pętla może je modyfikować.

- **Dobre praktyki:**

- Używaj zmiennych lokalnych w pętlach, gdy to możliwe, aby ograniczyć ich zakres i uniknąć błędów.

- Jeśli zmienna ma być używana w wielu pętlach lub funkcjach, zadeklaruj ją w odpowiednim zakresie (np. w `main()`), ale unikaj globalnych, chyba że są naprawdę potrzebne.
- W pętlach zagnieżdżonych upewnij się, że zmienne liczników mają unikalne nazwy (np. `i`, `j`, `k`), aby uniknąć konfliktów.

Lekcja 3

Temat: Funkcje

Funkcja to nazwany blok kodu, który wykonuje określone zadanie i może być wielokrotnie wywoływany w programie. Funkcje pozwalają na modularność, czytelność i ponowne wykorzystanie kodu. Składają się z:

- **Nagłówka** (określa nazwę, typ zwracany i parametry).
- **Ciała** (zawiera instrukcje do wykonania).

Funkcje mogą:

- **Zwracać wartość** (np. `int`, `double`, `std::string`) lub nie zwracać nic (`void`).
- **Przyjmować parametry** (dane wejściowe) lub działać bez nich.

Składnia:

```
typ_zwracany nazwa_funkcji(parametry) {  
    // Ciało funkcji  
    // Kod do wykonania  
    return wartość; // Jeśli funkcja zwraca wartość  
}
```

Przykład:

```
#include <iostream>
int dodaj(int a, int b) {
    return a + b;
}

int main() {
    int wynik = dodaj(3, 4);
    std::cout << "Wynik: " << wynik << std::endl;
    return 0;
}
```

Przekazywanie parametrów

a) Przekazywanie przez wartość

- ☐ Kopia argumentu jest przekazywana do funkcji.
- ☐ Zmiany w parametrze wewnątrz funkcji nie wpływają na oryginalną zmienną.
- ☐ Domyślny sposób przekazywania w C++.

```
#include <iostream>
void zwieksz(int x) {
    x++;
    std::cout << "W funkcji: " << x << std::endl;
}

int main() {
    int a = 5;
    zwieksz(a);
    std::cout << "Poza funkcją: " << a << std::endl;
    return 0;
}
```

Wynik:

W funkcji: 6

Poza funkcją: 5

b) Przekazywanie przez referencję

- ☐ Funkcja operuje na oryginalnej zmiennej poprzez jej referencję (alias).
- ☐ Używa się operatora & w deklaracji parametru.
- ☐ Zmiany w parametrze wpływają na oryginalną zmienną.
- ☐ Przydatne, gdy chcemy zmodyfikować argument lub uniknąć kopiowania dużych danych.

Przykład:

```
#include <iostream>
void zwieksz(int& x) {
    x++;
    std::cout << "W funkcji: " << x << std::endl;
}

int main() {
    int a = 5;
    zwieksz(a);
    std::cout << "Poza funkcją: " << a << std::endl;
    return 0;
}
```

Wynik:

W funkcji: 6

Poza funkcją: 6

c) Przekazywanie przez wskaźnik (adres pamięci zmiennej)

- ☐ Alternatywa dla referencji, używa wskaźników (*).
- ☐ Również pozwala modyfikować oryginalną zmienną, ale wymaga jawnego zarządzania adresami.

Przykład:

```

#include <iostream>
void zwieksz(int* x) {
    (*x)++;
    std::cout << "W funkcji: " << *x << std::endl;
}

int main() {
    int a = 5;
    zwieksz(&a);
    std::cout << "Poza funkcją: " << a << std::endl;
    return 0;
}

```

Wynik:

W funkcji: 6
Poza funkcją: 6

d) Domyślne parametry

Funkcje mogą mieć parametry z wartościami domyślnymi, które są używane, gdy argument nie zostanie podany.

Przykład:

```

int pomnoz(int a, int b = 2) {
    return a * b;
}

int main() {
    std::cout << pomnoz(5) << std::endl;
    std::cout << pomnoz(5, 3) << std::endl;
    return 0;
}

```

Lekcja 4

Temat: Tablice i łańcuchy znaków: Deklaracja tablic, operacje na tablicach, `std::string` i manipulacja ciągami.

Tablice to **zbiór elementów tego samego typu**. Każdy element ma swój **index**, zaczynający się od 0.

Deklaracja tablic:

Określa się: **typ elementów, nazwę tablicy i jej rozmiar** (stały w czasie kompilacji lub dynamiczny).

Składnia:

typ **nazwa_tablicy**[**rozmiar**];

Przykład:

```
int liczby[5]; // Tablica 5 liczb całkowitych
double oceny[10]; // Tablica 10 liczb zmiennoprzecinkowych
char znaki[3] = {'a', 'b', 'c'}; // Tablica znaków z inicjalizacją
int tablica[4] = {1, 2, 3, 4}; // Inicjalizacja wartości
```

Uwagi:

- **Rozmiar tablicy statycznej musi być znany w czasie kompilacji** (stała liczba lub wyrażenie stałe np.: `int liczby[3] = {1, 2, 3};`).
- **Brak inicjalizacji elementów tablicy powoduje, że zawierają one losowe wartości (dla typów prostych).**
- **Jeśli nie podasz wszystkich elementów, reszta zostanie zainicjalizowana wartością domyślną** (`" "` dla stringów, `0` dla typów liczbowych).

Podstawowe operacje na tablicy

```
#include <iostream>

int main() {

    int liczby[5] = { 10, 20, 30, 40, 50 };

    liczby[2] = 35;

    for (int i = 0; i < 5; i++) {

        std::cout << "Element " << i << ": " << liczby[i] << std::endl;

    }

    int suma = 0;

    for (int i = 0; i < 5; i++) {

        suma += liczby[i];

    }

    std::cout << "Suma: " << suma << std::endl;

    return 0;

}
```

Dodawanie i usuwanie elementów jest dostępne tylko w **kontenerach dynamicznych** (np. `std::vector`, `std::list`, `std::deque`).

std::vector - dynamiczna tablica

Najważniejsze operacje w `std::vector`:

- `push_back(x)` → dodaje element na końcu
- `pop_back()` → usuwa ostatni element
- `insert(iterator, x)` → wstawia element w dowolne miejsce
- `erase(iterator)` → usuwa element z dowolnego miejsca
- `clear()` → usuwa wszystkie elementy

```

#include <iostream>
#include <vector>

int main() {
    std::vector<int> liczby = {10, 20, 30, 40, 50};

    // Dodawanie elementów
    liczby.push_back(60);
    liczby.insert(liczby.begin() + 2, 15);

    std::cout << "Po dodaniu: ";
    for (int indexD : liczby) std::cout << indexD << " ";
    std::cout << "\n";

    // Usuwanie elementów
    liczby.pop_back();
    liczby.erase(liczby.begin() + 1);

    std::cout << "Po usunięciu: ";
    for (int indexU : liczby) std::cout << indexU << " ";
}

```

std::vector jest kontenerem ogólnym (szablonowym) i w STL (Standard Template Library) wszystkie operacje są zdefiniowane w sposób ujednolicony.

- **W C++ nie przyjmuje numeru indeksu jako liczby całkowitej (int), tylko iterator. Iterator działa jak wskaźnik i wskazuje na konkretne miejsce w kolekcji.**
- Dzięki temu ta sama składnia działa dla różnych kontenerów (std::vector, std::list, std::deque, itd.), nawet jeśli one nie mają dostępu do elementów po indeksie.

DO tego miejsca przerobione

std::list- lista dwukierunkowa (ang. doubly-linked list).

Najważniejsze operacje w std::list:

- **push_back(x) → dodaj na końcu**
- **push_front(x) → dodaj na początku**
- **insert(iterator, x) → wstaw element w środku**
- **pop_back() → usuń ostatni element**
- **pop_front() → usuń pierwszy element**
- **erase(iterator) → usuń element wskazywany przez iterator**

- **remove(value)** → **usuń wszystkie elementy o wartości value**
- **clear()** → **usuń wszystkie elementy**

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

```
int main() {
    std::list<int> liczby_list = { 10, 20, 30, 40, 50 };

    // Dodawanie elementów
    liczby_list.push_back(60);
    liczby_list.push_front(5);

    auto it = liczby_list.begin();
    std::advance(it, 2);
    liczby_list.insert(it, 15);

    std::cout << "Po dodaniu: ";
    for (int inD: liczby_list) std::cout << inD << " ";
    std::cout << "\n";

    // Usuwanie elementów
    liczby_list.pop_back();
    liczby_list.pop_front();
    it = liczby_list.begin();
    std::advance(it, 2);
    liczby_list.erase(it);

    std::cout << "Po usunięciu: ";
    for (int inU: liczby_list) std::cout << inU << " ";
    std::cout << "\n";
}
```

std::deque - (double-ended queue – kolejka dwustronna)

```
#include <iostream>
#include <deque>
```

```
int main() {
    std::deque<int> liczby_deque = { 10, 20, 30, 40, 50 };

    liczby_deque.push_back(60);
```



```

    liczby_deque.push_front(5);

    std::cout << "Po dodaniu: ";
    for (int i5 : liczby_deque) std::cout << i5 << " ";
    std::cout << "\n";

    liczby_deque.pop_back();
    liczby_deque.pop_front();

    std::cout << "Po usunięciu: ";
    for (int i6 : liczby_deque) std::cout << i6 << " ";
    std::cout << "\n";

    auto it2 = liczby_deque.begin();
    std::advance(it2, 2);
    it2 = liczby_deque.insert(it2, 99);
    liczby_deque.erase(it2);

    std::cout << "Po wstawianiu i usuwaniu w środku: ";
    for (int i7 : liczby_deque) std::cout << i7 << " ";
    std::cout << "\n";

}

```

Najważniejsze operacje w `std::deque`:

- `push_back(x)` → **dodaje element na końcu**
- `push_front(x)` → **dodaje element na początku**
- `pop_back()` → **usuwa element z końca**
- `pop_front()` → **usuwa element z początku**
- `insert(iterator, x)` → **wstawia element w dowolnym miejscu**
- `erase(iterator)` → **usuwa element w dowolnym miejscu**
- `clear()` → **usuwa wszystkie elementy**

Przekazywanie tablicy do funkcji:

```

#include <iostream>
void zwieksz(int* tab, int rozmiar) {
    for (int i = 0; i < rozmiar; i++) {
        tab[i]++; // Modyfikacja elementów
    }
}

```

```
int main() {
    int liczby[3] = {1, 2, 3};
    zwieksz(liczby, 3);
    for (int i = 0; i < 3; i++) {
        std::cout << liczby[i] << " ";
    }
    return 0;
}
```

Uwaga: Przy przekazywaniu tablicy do funkcji należy podać jej rozmiar, bo tablica w funkcji "traci" informacje o swoim rozmiarze.

Tablice dynamiczne

Jeśli rozmiar tablicy nie jest znany w czasie kompilacji, można użyć dynamicznej alokacji pamięci (**new** i **delete**).

Przykład

```
#include <iostream>
int main() {
    int rozmiar;
    std::cout << "Podaj rozmiar tablicy: ";
    std::cin >> rozmiar;

    // Alokacja dynamiczna
    int* tab = new int[rozmiar];

    // Wypełnienie tablicy
    for (int i = 0; i < rozmiar; i++) {
        tab[i] = i + 1;
    }

    // Wypisanie
    for (int i = 0; i < rozmiar; i++) {
        std::cout << tab[i] << " ";
    }

    // Zwolnienie pamięci
    delete[] tab;
```

```
    return 0;
}
```

Uwaga: Dynamicznie zaalokowaną pamięć trzeba zwolnić (`delete[]`), aby uniknąć wycieków pamięci.

Łańcuchy znaków

W C++ łańcuchy znaków można reprezentować na dwa sposoby:

1. **Tablice znaków w stylu C** (`char[]`) – tradycyjne, zakończone znakiem `'\0'` (null-terminator).
2. **Klasa `std::string`** – nowoczesny sposób, wygodniejszy i bezpieczniejszy.

Tablice znaków

Deklaracja:

Tablica znaków to tablica typu `char`, zakończona znakiem `'\0'`, który oznacza koniec ciągu.

```
char tekst[] = "Witaj"; // Automatycznie dodaje '\0'
char tekst2[6] = {'W', 'i', 't', 'a', 'j', '\0'};
```

Operacje:

- Dostęp do znaków: `tekst[indeks]`.
- Modyfikacja: `tekst[indeks] = 'x'`.
- Funkcje z biblioteki `<cstring>`:
 - `strlen(tekst)` – długość ciągu.
 - `strcpy(dest, src)` – kopiowanie ciągu.
 - `strcmp(s1, s2)` – porównywanie ciągów.
 - `strcat(dest, src)` – konkatencja.

Przykład:

```
#include <iostream>
#include <cstring>
int main() {
    char tekst[] = "Witaj";

    // Długość ciągu
    std::cout << "Długość: " << strlen(tekst) << std::endl; // Wypisze: 5
}
```

```

// Kopiowanie
char kopia[10];
strcpy(kopia, tekst);
std::cout << "Kopia: " << kopia << std::endl; // Wypisze: Witaj

// Konkatenacja
strcat(kopia, "!");
std::cout << "Po konkatenacji: " << kopia << std::endl; // Wypisze: Witaj!

return 0;
}

```

Wady:

- Ryzyko błędów (np. przepełnienie bufora).
- Ręczna obsługa pamięci.
- Konieczność pamiętania o '\0'.

Klasa `std::string`

Czym jest `std::string`?

`std::string` to klasa z biblioteki standardowej (`<string>`), która ułatwia manipulację ciągami znaków. Jest bezpieczniejsza i bardziej funkcjonalna niż tablice znaków.

Deklaracja:

```

#include <string>
std::string tekst = "Witaj";
std::string tekst2("C++");

```

Podstawowe operacje:

- **Dostęp do znaków:** `tekst[indeks]` lub `tekst.at(indeks)` (z sprawdzaniem zakresu).
- **Długość:** `tekst.length()` lub `tekst.size()`.
- **Konkatenacja:** Operator `+` lub `+=`.
- **Porównywanie:** Operatory `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Podciąg:** `tekst.substr(pozycja, długość)`.

- **Wyszukiwanie:** tekst.find(ciąg) – zwraca pozycję ciągu lub std::string::npos jeśli nie znaleziono.
- **Zamiana:** tekst.replace(pozycja, długość, nowy_ciąg).

Przykład: Manipulacja std::string:

```
#include <iostream>
#include <string>
int main() {
    std::string tekst = "Witaj, C++!";

    // Długość
    std::cout << "Długość: " << tekst.length() << std::endl; // Wypisze: 11

    // Konkatenacja
    tekst += " Jest super!";
    std::cout << "Po konkatenacji: " << tekst << std::endl; // Wypisze: Witaj, C++!
    Jest super!

    // Podciąg
    std::string podciąg = tekst.substr(7, 3); // Zaczyna od pozycji 7, bierze 3 znaki
    std::cout << "Podciąg: " << podciąg << std::endl; // Wypisze: C++

    // Wyszukiwanie
    size_t pozycja = tekst.find("C++");
    if (pozycja != std::string::npos) {
        std::cout << "Znaleziono 'C++' na pozycji: " << pozycja << std::endl; //
    Wypisze: 7
    }

    // Zamiana
    tekst.replace(7, 3, "Python");
    std::cout << "Po zamianie: " << tekst << std::endl; // Wypisze: Witaj, Python!
    Jest super!

    return 0;
}
```

Zalety std::string:

- Automatyczne zarządzanie pamięcią.
- Bezpieczeństwo (brak ryzyka przepełnienia bufora).
- Bogaty zestaw metod do manipulacji.

- Łatwe porównywanie i konkatencja.

Zastosowania `std::string`:

- Przetwarzanie tekstu (np. parsowanie danych, formatowanie).
- Wczytywanie danych od użytkownika (np. z `std::cin`).
- Operacje na danych tekstowych w aplikacjach (np. wyszukiwanie, zamiana).