

# **PRZEDMIOT: Podstawy programowania**

KLASA: 2A gr. 2

## **Lekcja 1,2**

**Temat:** Język niskiego poziomu i wysokiego poziomu.  
Operacje wejścia/wyjścia. Typy danych

### ♦ **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.

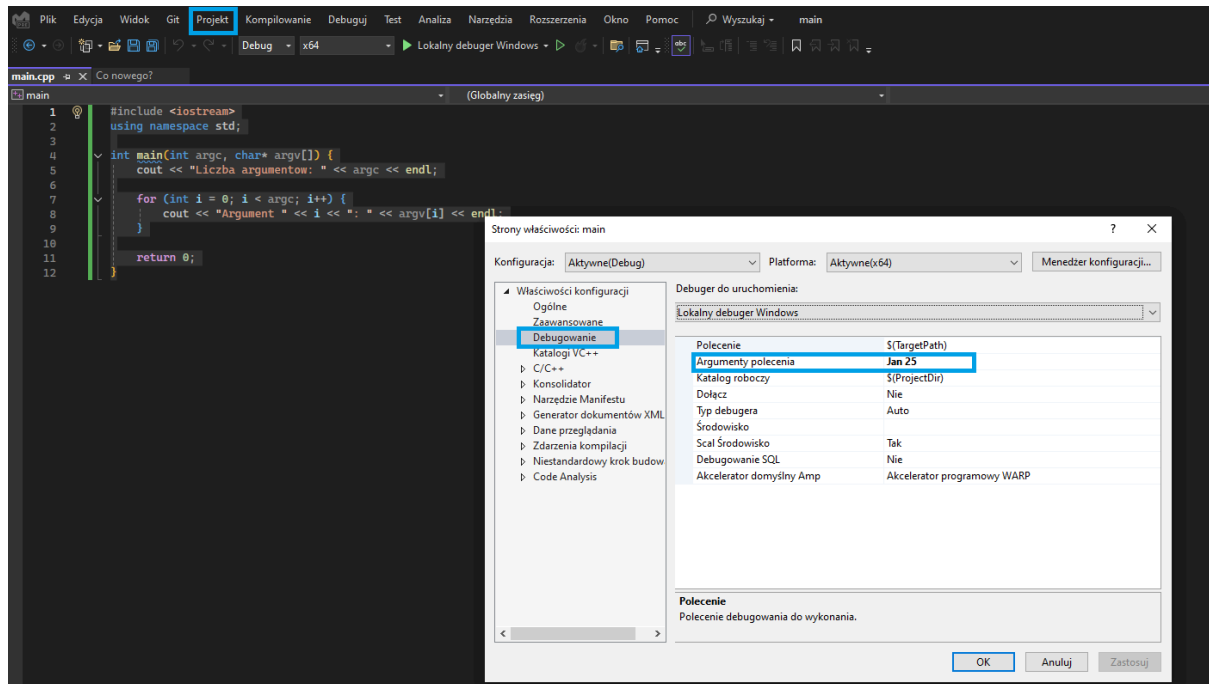
##### 2. Zwracanie wartości typu `int`

- `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	10000000000
<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>&lt;string&gt;</code> )	Ciąg znaków	"Hello"

### ♦ 3. Typy wskaźnikowe i referencje

Typ	Opis	Przykład
<code>int*</code>	Wskaźnik na <code>int</code>	<code>int* ptr = &amp;x;</code>
<code>double*</code>	Wskaźnik na <code>double</code>	<code>double* dp;</code>
<code>int&amp;</code>	Referencja (alias) do zmiennej	<code>int&amp; 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 3

## Temat: Instrukcje warunkowe

### ♦ 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;
```

```
}
```

# Lekcja 4

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

## 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 5

### 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 6

**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, która przechowuje elementy w ciągłym bloku pamięci (jak klasyczna tablica C-style, ale z automatycznym zarządzaniem rozmiarem).

## 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::cout << "\n";
}
```

**`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.

**std::list**- to dwukierunkowa lista powiązana (ang. doubly-linked list). w której każdy element przechowuje wskaźniki do poprzedniego i następnego elementu.

### **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** - kolejka dwustronna (ang. double-ended queue), która pozwala na szybkie dodawanie i usuwanie elementów na obu końcach.

```
#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

    // Konkatencja
    strcat(kopia, "!");
    std::cout << "Po konkatencji: " << 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 podciag = tekst.substr(7, 3); // Zaczyna od pozycji 7, bierze 3 znaki
    std::cout << "Podciąg: " << podciag << 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).

## Lekcja 7

### Temat: Funkcja rekurencyjna. Stos wywołań

Funkcje rekurencyjne w C++ to funkcje, **które wywołują same siebie w celu rozwiązania problemu poprzez rozbicie go na mniejsze, podobne podproblemy.**

Rekurencja jest techniką programistyczną, w której funkcja wywołuje się z nowymi parametrami, aż osiągnie warunek bazowy (przypadek bazowy), który kończy rekurencję i pozwala na zwrócenie wyniku.

Jak działają funkcje rekurencyjne?

#### 1. Warunek bazowy (base case):

- To warunek, który zatrzymuje rekurencję, zapobiegając nieskończonym wywołaniom.
- Bez warunku bazowego funkcja może spowodować przepełnienie stosu (stack overflow).
- Przykład: W obliczaniu silni, warunkiem bazowym może być `n == 0` lub `n == 1`.

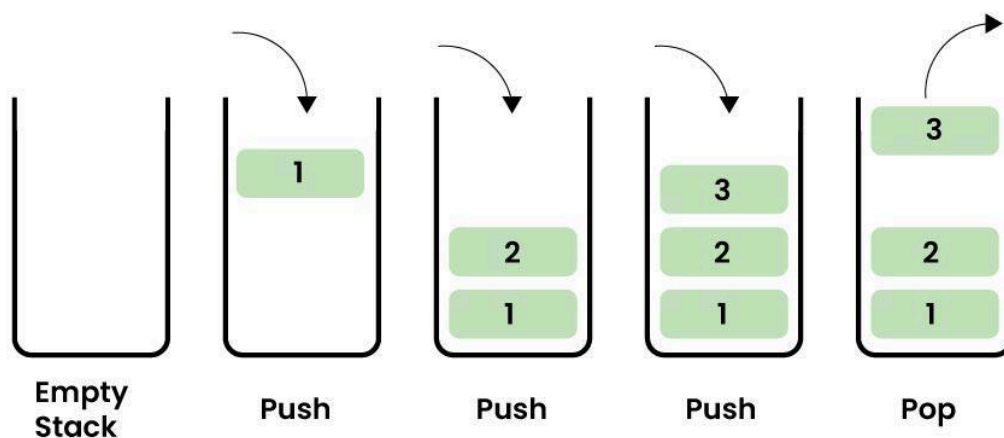
## 2. Krok rekurencyjny (recursive case):

- To część funkcji, w której wywołuje ona samą siebie z parametrem bliższym warunkowi bazowemu.
- Każde wywołanie rekurencyjne powinno zmniejszać rozmiar problemu, np. zmniejszać wartość parametru.

## 3. Stos wywołań (call stack):

- Każde wywołanie funkcji rekurencyjnej jest zapisywane na stosie wywołań.
- Po osiągnięciu warunku bazowego, funkcja zaczyna "zwijać" stos, zwracając wyniki od najgłębszego wywołania do pierwszego.

**Stos wywołań (call stack) w C++ to struktura danych w pamięci komputera, która zarządza wywołaniami funkcji** w trakcie wykonywania programu. Jest to **stos (ang. stack), czyli struktura typu LIFO (Last In, First Out – ostatnie na wejściu, pierwsze na wyjściu)**, używana do przechowywania informacji o aktywnych funkcjach, ich parametrach, zmiennych lokalnych oraz miejscu powrotu po zakończeniu funkcji. W kontekście funkcji rekurencyjnych stos wywołań odgrywa kluczową rolę, ponieważ **każde rekurencyjne wywołanie funkcji dodaje nowy rekord na stosie**.



LIFO Operations in stack



## Przykład zastosowania funkcji rekurencyjnej w C++ Obliczanie silni

Silnia liczby  $n$  (oznaczona jako  $n!$ ) to iloczyn wszystkich dodatnich liczb całkowitych mniejszych lub równych  $n$ . Definicja rekurencyjna:

- $n! = n \times (n-1)!$
- Warunek bazowy:  $0! = 1$  lub  $1! = 1$

```

#include <iostream>
using namespace std;

unsigned long long factorial(int n) {
    // Warunek bazowy
    if (n == 0 || n == 1) {
        return 1;
    }
    // Krok rekurencyjny
    return n * factorial(n - 1);
}

int main() {
    int n;
    cout << "Podaj liczbe n: ";
    cin >> n;

    if (n < 0) {
        cout << "Silnia nie jest zdefiniowana dla liczb ujemnych!" << endl;
    } else {
        cout << "Silnia z " << n << " wynosi: " << factorial(n) << endl;
    }

    return 0;
}

```

### Przykładowe działanie dla n=4

- $\text{factorial}(4) = 4 \times \text{factorial}(3)$
- $\text{factorial}(3) = 3 \times \text{factorial}(2)$
- $\text{factorial}(2) = 2 \times \text{factorial}(1)$
- $\text{factorial}(1) = 1$  (warunek bazowy)

### Wstecz

- **$\text{factorial}(1) = 1 = 1$ ,**
- **$\text{factorial}(2) = 2 \times 1 = 2$**
- **$\text{factorial}(3) = 3 \times 2 = 6$ ,**
- **$\text{factorial}(4) = 4 \times 6 = 24$**

## Wynik w konsoli:

Podaj liczbę n: 4

Silnia z 4 wynosi: 24

### 1. Struktura stosu wywołań

- **Stos:** To obszar pamięci przydzielany dla każdego wątku programu, zarządzany automatycznie przez system operacyjny i kompilator.
- **Ramka stosu (*stack frame* lub *activation record*):** Każdy rekord na stosie przechowuje informacje o jednym wywołaniu funkcji, takie jak:
  - **Adres powrotu:** Miejsce w kodzie, do którego program wróci po zakończeniu funkcji.
  - **Parametry funkcji:** Wartości przekazane do funkcji.
  - **Zmienne lokalne:** Zmienne zadeklarowane wewnątrz funkcji.
  - **Rejestry:** Czasami zapisywane są wartości rejestrów procesora (np. licznik programu).
- **Gdy funkcja jest wywoływana, nowa ramka stosu jest "pushowana" (dodawana) na wierzch stosu. Gdy funkcja się kończy, ramka jest "popowana" (usuwana), a program wraca do poprzedniego miejsca wywołania.**

### 2. Przepelnienie stosu (*stack overflow*)

- **Stos ma ograniczony rozmiar** (zazwyczaj kilka MB, zależnie od systemu i ustawień).
- **Zbyt wiele wywołań funkcji (np. głęboka rekurencja) może wypełnić stos, powodując błąd *stack overflow*.**
- **Przykład: Rekurencyjne wywołanie funkcji bez warunku bazowego**

## Lekcja 8



# **Temat:** Struktury (structs) i unie (unions): Definiowanie struktur i jej pól. Definiowanie unii i jej pól. Struktury i unie zagnieżdżone. std::variant

## **Struktury (structs) w C++**

Struktury w C++ to mechanizm **pozwalający na grupowanie różnych typów danych w jedną nazwę**, co ułatwia organizację kodu. **Struktura jest zbiorem pól** (członków), które **mogą być różnych typów**, takich jak liczby, ciągi znaków czy wskaźniki. Struktury są podstawowym narzędziem do tworzenia własnych typów danych, np. reprezentujących obiekty z realnego świata.

### **Definiowanie struktury i jej pól**

Strukturę definiujemy za pomocą słowa kluczowego **struct**, a następnie podajemy **nazwę i listę pól** w nawiasach kwadratowych **{}**. Pola to zmienne wewnątrz struktury, które mają swoje typy i nazwy.

### **Przykład definicji struktury:**

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

struct Osoba {
    string imie;    // Pole typu string
    int wiek;       // Pole typu int
    double wzrost;  // Pole typu double
};

// Aby użyć struktury, deklarujemy zmienną o tym typie i inicjalizujemy pola:

int main() {
    Osoba jan;      // Deklaracja zmiennej struktury
```

```

jan.imie = "Jan";    // Dostęp do pola przez operator kropki (.)
jan.wiek = 30;
jan.wzrost = 1.75;

cout << jan.imie << " ma " << jan.wiek << " lat i wzrost " << jan.wzrost << "
m." << endl;
return 0;
}

```

**Wynik wykonania:** Jan ma 30 lat i wzrost 1.75 m.

## Struktury zagnieżdżone

**Struktury mogą zawierać inne struktury jako pola**, co pozwala na hierarchiczną organizację danych. To przydatne w złożonych modelach, np. adres w strukturze osoby.

### Przykład struktury zagnieżdżonej:

```

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

struct Adres {
    string miasto;
    string ulica;
    int numerDomu;
};

struct Osoba {
    string imie;
    int wiek;
    Adres adres; // Zagnieżdżona struktura Adres
};

int main() {
    Osoba anna;
    anna.imie = "Anna";
    anna.wiek = 25;
    anna.adres.miasto = "Warszawa"; // Dostęp do zagnieżdżonego pola
    anna.adres.ulica = "Marszałkowska";
}

```

```

anna.adres.numerDomu = 10;

cout << anna.imie << " mieszka w " << anna.adres.miasto << endl;
return 0;
}

```

**Wynik wykonania:** Anna mieszka w Warszawa

**Zastosowanie:** W systemach informatycznych, np. struktura pracownika z zagnieżdżoną strukturą działu (Dział { nazwa, szef }) lub w grafice komputerowej (obiekt 3D z podstrukturą transformacji: pozycja, rotacja).

## Unie (**unions**) w C++

Unie to podobny mechanizm do struktur, ale z kluczową różnicą: **wszystkie pola unii zajmują tę samą pamięć**. W dowolnym momencie aktywny jest tylko jeden z pól (ten ostatni zainicjalizowany), a rozmiar unii to maksymalny rozmiar jej największego pola. Unie oszczędzają pamięć, ale wymagają ostrożności, by nie odczytywać nieaktywnego pola (może prowadzić do błędów).

### Definiowanie unii i jej pól

Unię definiujemy słowem kluczowym **union**, a pola definiujemy podobnie jak w strukturach.

#### Przykład definicji unii:

```

#include <iostream>
using namespace std;

union Liczba {
    int calkowita;           // Pole typu int (4 bajty)
    double ulamkowa;        // Pole typu double (8 bajtów)
    // Rozmiar unii: 8 bajtów (max z pól)
};

// Użycie (pamiętaj: tylko jedno pole jest aktywne!):

int main() {
    Liczba x;
    x.calkowita = 42;        // Aktywne pole: calkowita
    cout << "Całkowita: " << x.calkowita << endl;
}

```

```

x.ulamkowa = 3.14;           // Teraz aktywne: ulamkowa (nadpisuje pamięć!)
cout << "Ułamkowa: " << x.ulamkowa << endl;
// UWAGA: x.calkowita jest teraz nieokreślone!

return 0;
}

```

### Wynik wykonania:

Całkowita: 42

Ułamkowa: 3.14

**Zastosowanie: Unie są używane do optymalizacji pamięci, np. w parserach** (reprezentacja wariantów danych: liczba lub tekst w tej samej pamięci) czy w systemach wbudowanych (tagowane unie z enum do rozróżniania typów, np. w protokołach komunikacyjnych).

### Unie z strukturami (zagnieżdżone)

Można zagnieżdżać unie w strukturach lub odwrotnie, co pozwala na elastyczne typy wariantowe (variant types).

#### Przykład unii zagnieżdżonej w strukturze (tagowana unia):

```

#include <iostream>
using namespace std;

enum TypDanych { CALKOWITA, ULAMKOWA }; // Enum do tagowania

struct Dane {
    TypDanych typ;           // Tag wskazujący aktywne pole
    union {                  // Zagnieżdżona unia
        int calkowita;
        double ulamkowa;
    };
};

int main() {

    Dane y;
    y.typ = CALKOWITA;
}

```

```

y.calkowita = 100;

if (y.typ == CALKOWITA) {
    cout << "Wartość: " << y.calkowita << endl;
}

y.typ = ULAMKOWA;
y.ulamkowa = 2.718;
if (y.typ == ULAMKOWA) {
    cout << "Wartość: " << y.ulamkowa << endl;
}

return 0;
}

```

### Wynik wykonania:

Wartość: 100

Wartość: 2.718

## std::variant

**std::variant** to kontener typów wariantowych wprowadzony w standardzie C++17 (w nagłówku `<variant>`). Pozwala on na przechowywanie **jednej wartości z kilku różnych typów** w jednej zmiennej, z pełną kontrolą typów w czasie kompilacji. Jest to bezpieczniejsza i bardziej nowoczesna alternatywa dla tradycyjnych unii (union), które nie sprawdzają typów w runtime i mogą prowadzić do błędów. **std::variant automatycznie śledzi, który typ jest aktywny** (używa wewnętrznego "tagu" indeksu), i zapewnia mechanizmy do dostępu do wartości bez ryzyka nieokreślonych zachowań.

### Kluczowe cechy

- **Typy wariantowe:** Określasz listę dozwolonych typów w szablonie, np. **std::variant<int, std::string, double>**.
- **Aktywny typ:** **Tylko jeden typ jest aktywny na raz**; próba dostępu do nieaktywnego powoduje wyjątek `std::bad_variant_access`.
- **Rozmiar:** Rozmiar wariantu to maksymalny rozmiar jego typów plus overhead (zwykle 1 bajt na tag).
- **Nieinwazyjny:** Nie modyfikuje typów – to po prostu "pudełko" na jeden z nich.

- **Brak null state:** W przeciwieństwie do `std::optional`, wariant zawsze ma wartość (chyba że użyjesz `std::monostate` jako pierwszego typu dla "pustego" stanu).

## Definiowanie i inicjalizacja

Aby użyć `std::variant`, dołącz `#include <variant>` i `#include <iostream>` (do przykładów). Inicjalizujesz go przez podanie wartości dla konkretnego typu.

### Przykład podstawowy:

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

int main() {
    // Definicja wariantu z trzema typami
    std::variant<int, std::string, double> v;

    v = 42;                // Aktywny typ: int
    v = std::string("Witaj"); // Aktywny typ: std::string
    v = 3.14;              // Aktywny typ: double

    // Inicjalizacja w konstruktorze
    std::variant<int, std::string, double> w = "C++17";
    std::variant<int, std::string, double> x{42};

    std::cout << "Wartość w v: " << std::get<double>(v) << std::endl; // 3.14
    (jeśli ostatni)
    return 0;
}
```

**Wynik wykonania:** Wartość w v: 3.14

**Uwaga:** Użyj `std::get<T>(variant)` do dostępu, gdzie T to oczekiwany typ. Jeśli typ nieaktywny – wyjątek!

### Dostęp do wartości

- **`std::get<T>(v)`:** Zwraca referencję do wartości typu T. Bezpieczne tylko jeśli T jest aktywny.
- **`std::get<indeks>(v)`:** Dostęp po indeksie (0 dla pierwszego typu).

- **std::holds\_alternative<T>(v)**: Sprawdza, czy aktywny jest typ T (zwraca bool).
- **v.index()**: Zwraca indeks aktywnego typu (size\_t).

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

```
int main() {
```

```
    std::variant<int, std::string, double> v = 100;
```

```
    if (std::holds_alternative<int>(v)) {
        std::cout << "To int: " << std::get<int>(v) << std::endl;
    } else if (std::holds_alternative<std::string>(v)) {
        std::cout << "To string: " << std::get<std::string>(v) << std::endl;
    }
}
```

```
    std::cout << "Indeks: " << v.index() << std::endl;
```

```
    return 0;
}
```

### Wynik wykonania:

To int: 100

Indeks: 0

### Przetwarzanie wartości: std::visit

Najlepszy sposób na radzenie sobie z różnymi typami w std::variant to użycie funkcji std::visit. Ona automatycznie "wybiera" i uruchamia odpowiednią część kodu (np. lambda lub funkcję) dla tego typu, który jest akurat aktywny w wariancie. Dzięki temu nie musisz pisać długiego łańcucha poleceń if-else (sprawdzaj typ, potem inny, potem jeszcze inny).

### Przykład z std::visit:

```

#include <iostream>
#include <variant>
#include <string>

int main() {

    std::variant<int, std::string, double> v = 42.5;

    // Visitor jako lambda
    std::visit([](auto&& arg) {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, int>) {
            std::cout << "Liczba całkowita: " << arg << std::endl;
        } else if constexpr (std::is_same_v<T, std::string>) {
            std::cout << "Tekst: " << arg << std::endl;
        } else if constexpr (std::is_same_v<T, double>) {
            std::cout << "Liczba zmiennoprzecinkowa: " << arg << std::endl;
        }
    }, v); // Wywołania dla double

    return 0;
}

```

## Lekcja 9

**Temat:** Klasy i obiekty: Podstawy OOP – klasy, konstruktory, destruktory, metody.

**Programowanie obiektowe (OOP) w C++ to paradygmat, który pozwala na modelowanie świata rzeczywistego poprzez klasy i obiekty.** Kluczowe koncepcje OOP to **inkapsulacja** (ukrywanie szczegółów implementacji), **dziedziczenie** (wykorzystywanie kodu z klas bazowych) i **polimorfizm** (różne zachowania dla podobnych interfejsów).

### 1. Klasy i obiekty



**Klasa** to szablon definiujący strukturę i zachowanie obiektów. Definiuje ona pola (dane) i metody (funkcje). **Obiekt** to instancja klasy – konkretny "egzemplarz" z własnymi danymi.

- **Zastosowanie:** Klasy pozwalają organizować kod w logiczne jednostki. Na przykład, w aplikacji bankowej klasa Konto może przechowywać saldo i umożliwiać operacje, a każdy klient ma swój obiekt Konto.

**Przykład kodu** (prosta klasa Konto):

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

class Konto {
private: // Inkapsulacja: dane ukryte
    std::string wlasciciel;
    double saldo;

public: // Interfejs publiczny
    // Konstruktor – omówiony poniżej
    Konto(std::string nazwa, double poczatkowe_saldo) {
        wlasciciel = nazwa;
        saldo = poczatkowe_saldo;
    }

    // Metoda – omówiona poniżej
    void wplata(double kwota) {
        if (kwota > 0) {
            saldo += kwota;
            std::cout << "Wplata: " << kwota << std::endl;
        }
    }

    double pobierz_saldo() const { // Metoda dostępu (getter)
        return saldo;
    }
};

int main() {
    // Tworzenie obiektów (instancji klasy)
    Konto konto1("Jan Kowalski", 1000.0); // Obiekt konto1
    Konto konto2("Anna Nowak", 500.0);    // Obiekt konto2

    konto1.wplata(200.0); // Użycie metody na obiekcie
    std::cout << "Saldo konto1: " << konto1.pobierz_saldo() << std::endl; //
    Wyjście: 1200
```

```
    return 0;
}
```

W tym przykładzie:

- Klasa Konto definiuje strukturę.
- Obiekty konto1 i konto2 to niezależne instancje z własnymi danymi (saldo).
- Zastosowanie: W większym systemie bankowym, każdy klient ma obiekt Konto, co ułatwia zarządzanie tysiącami kont bez duplikowania kodu.

## 2. Konstruktory

**Konstruktor** to specjalna metoda wywoływana automatycznie przy tworzeniu obiektu. Inicjalizuje pola klasy. Może być domyślny (bez argumentów), parametryzowany lub kopiujący. Jeśli nie zdefiniujesz konstruktora, kompilator utworzy domyślny.

- **Zastosowanie:** Zapewniają poprawne zainicjowanie obiektu, np. ustawienie początkowego salda w koncie bankowym, co zapobiega błędom (jak niezerowe saldo).

**Przykład** (w powyższym kodzie widać parametryzowany konstruktor). Dodajmy domyślny:

```
class Konto {
private:
    std::string wlasciciel;
    double saldo;

public:
    // Domyślny konstruktor
    Konto() : wlasciciel("Nieznany"), saldo(0.0) {} // Inicjalizacja listą

    // Parametryzowany konstruktor
    Konto(std::string nazwa, double poczatkowe_saldo) : wlasciciel(nazwa),
    saldo(poczatkowe_saldo) {}

    // ... reszta metod
};

int main() {
    Konto konto3; // Używa domyślnego konstruktora: saldo = 0
    std::cout << "Saldo konto3: " << konto3.pobierz_saldo() << std::endl; //
    Wyjście: 0
    return 0;
}
```

Zastosowanie: W grze komputerowej konstruktor klasy Postac może ustawić początkowe zdrowie i poziom gracza przy starcie gry.

### 3. Destruktory

**Destruktor** to specjalna metoda wywoływana automatycznie przy niszczeniu obiektu (np. wyjście z zakresu, delete). Oznacza się `~NazwaKlasy()`. Służy do zwalniania zasobów (pamięci, plików).

- **Zastosowanie:** Zapobiegają wyciekom pamięci (memory leaks). Np. w klasie zarządzającej plikiem, destruktory zamyka plik automatycznie.

**Przykład** (klasa z dynamiczną pamięcią):

```
#include <iostream>

class Lista {
private:
    int* dane; // Wskaźnik do dynamicznej tablicy
    int rozmiar;

public:
    Lista(int n) : rozmiar(n) {
        dane = new int[n]; // Alokacja pamięci
        for (int i = 0; i < n; ++i) dane[i] = i;
    }

    ~Lista() { // Destruktor
        delete[] dane; // Zwolnienie pamięci
        std::cout << "Destruktor: Pamięć zwolniona!" << std::endl;
    }

    void wyswietl() {
        for (int i = 0; i < rozmiar; ++i) {
            std::cout << dane[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    {
        Lista moja_lista(5); // Tworzenie obiektu
        moja_lista.wyswietl(); // Wyjście: 0 1 2 3 4
    } // Tutaj obiekt wychodzi z zakresu – wywołuje destruktora
    // Wyjście: Destruktor: Pamięć zwolniona!
    return 0;
}
```

## 4. Metody

**Metody** to funkcje należące do klasy. Mogą być dostępne publicznie (public), prywatnie (private) lub chronione (protected). Metody stałe (const) nie modyfikują obiektu.

- **Zastosowanie:** Definiują zachowanie obiektu, np. metody w klasie Samochod mogą symulować jazdę lub tankowanie.

**Przykład** (rozszerzając klasę Konto o metodę wypłaty):

```
class Konto {
    // ... pola i konstruktory jak wyżej

public:
    void wypłata(double kwota) {
        if (kwota > 0 && kwota <= saldo) {
            saldo -= kwota;
            std::cout << "Wypłata: " << kwota << std::endl;
        } else {
            std::cout << "Błąd: Niewystarczające środki!" << std::endl;
        }
    }

    // Metoda stała (nie zmienia obiektu)
    std::string pobierz_wlasciciela() const {
        return wlasciciel;
    }
};

int main() {
    Konto konto("Jan", 1000.0);
    konto.wypłata(300.0); // Wyjście: Wypłata: 300
    std::cout << "Właściciel: " << konto.pobierz_wlasciciela() << std::endl;
    return 0;
}
```

Zastosowanie: W symulacji gry, metody klasy Gracz mogą obsługiwać ruchy (np. idźNaprzód()), co czyni kod modularnym i łatwym do rozszerzenia.

## Podsumowanie zastosowań

Element OOP	Przykładowe zastosowanie	Korzyść
<b>Klasa i obiekt</b>	Modelowanie kont bankowych	Reużywalność kodu dla wielu instancji
<b>Konstruktor</b>	Inicjalizacja postaci w grze	Zapobieganie błędom początkowym

<b>Destruktor</b>	Zamykanie plików w edytorze tekstu	Automatyczne zarządzanie zasobami
<b>Metody</b>	Operacje na bazie danych (CRUD)	Enkapsulacja logiki biznesowej

## Lekcja 10

**Temat:** Dziedziczenie pojedyncze i wielokrotne, klasy bazowe i pochodne, nadpisywanie metod.

**Dziedziczenie to mechanizm, który pozwala na tworzenie nowych klas na podstawie istniejących.**

Umożliwia to ponowne wykorzystanie kodu, rozszerzanie funkcjonalności i budowanie hierarchii klas.

### 1. Klasy bazowe i pochodne

- **Klasa bazowa** (base class): **To klasa, z której dziedziczymy. Zawiera wspólne cechy i metody, które mogą być udostępnione klasom pochodnym.** Nazywana też klasą nadrzędną (parent class) lub superklasą.
- **Klasa pochodna** (derived class): **To klasa, która dziedziczy z klasy bazowej. Może dodawać nowe pola, metody lub modyfikować istniejące.** Nazywana też klasą potomną (child class) lub podklasą.

Dziedziczenie definiujemy za pomocą składni : public NazwaKlasyBazowej (lub protected/private, ale najczęściej używa się public dla dostępu do elementów publicznych).

### Przykład:

```
#include <iostream>

class Zwierze {
public:
    void jedz() {
        std::cout << "Zwierze je." << std::endl;
    }
}
```

```
};

class Pies : public Zwierze {
public:
    void szczekaj() {
        std::cout << "Pies szczeka: Hau!" << std::endl;
    }
};

int main() {
    Pies mojPies;
    mojPies.jedz();
    mojPies.szczekaj();
    return 0;
}
```

W tym przykładzie *Pies* dziedziczy z *Zwierze*, więc ma dostęp do metody *jedz()*.

## 2. Dziedziczenie pojedyncze

Dziedziczenie pojedyncze oznacza, że klasa pochodna dziedziczy z **dokładnie jednej** klasy bazowej. To najprostsza i najbezpieczniejsza forma dziedziczenia w C++, unikająca komplikacji jak konfliktów nazw.

### Przykład:

```
class Pojazd {
public:
    void jedz() {
        std::cout << "Pojazd jedzie." << std::endl;
    }
};

class Samochod : public Pojazd {
public:
    void trąb() {
        std::cout << "Samochod trąbi: Beep!" << std::endl;
    }
};

int main() {
    Samochod mojSamochod;
    mojSamochod.jedz();
    mojSamochod.trąb();
    return 0;
}
```

Tutaj *Samochod* dziedziczy tylko z *Pojazd*.

## 3. Dziedziczenie wielokrotne

Dziedziczenie wielokrotne pozwala klasie pochodnej dziedziczyć z **więcej niż jednej** klasy bazowej. Jest to unikalne dla C++ (w porównaniu do np. Javy, gdzie jest zabronione). **Jednak może prowadzić do problemów**, takich jak "**diament śmierci**" (diamond problem), **gdy dwie klasy bazowe mają wspólną bazę**, co **powoduje duplikację dziedziczenia**. Aby to rozwiązać, używa się dziedziczenia **wirtualnego (virtual)**.

### Przykład:

```
class Silnik {
public:
    void uruchom() {
        std::cout << "Silnik uruchomiony." << std::endl;
    }
};

class Kola {
public:
    void tocz() {
        std::cout << "Kola sie tocza." << std::endl;
    }
};

class Samochod : public Silnik, public Kola {
public:
    void jedz() {
        uruchom();
        tocz();
        std::cout << "Samochod jedzie." << std::endl;
    }
};

int main() {
    Samochod mojSamochod;
    mojSamochod.jedz();
    return 0;
}
```

### Przykład z problemem diamentu i rozwiązaniem:

```
#include <iostream>

class Zwierze {
public:
    void jedz() {
        std::cout << "Zwierze je." << std::endl;
    }
};

class Ptak : public Zwierze {}; // Bez virtual, aby nie było błędu należy dodać virtual
```

```
class Ssaki : public Zwierze {}; // Bez virtual, aby nie było błędu należy dodać virtual
```

```
class Nietoperz : public Ptak, public Ssaki {}; // Problem: dwie kopie Zwierze
```

```
int main() {  
    Nietoperz mojNietoperz;  
    mojNietoperz.jedz(); // Tutaj błąd kompilacji: ambiguous call to 'jedz()'  
    return 0;  
}
```

Bez virtual kompilator zgłosi błąd przy dostępie do jedz() w Nietoperz, bo nie wie, którą wersję wybrać. Z virtual dziedziczenie jest współdzielone.

#### 4. Nadpisywanie metod (method overriding)

Nadpisywanie to mechanizm, w którym klasa pochodna definiuje na nowo metodę z klasy bazowej, zmieniając jej implementację. Aby to działało:

- Metoda w klasie bazowej musi być oznaczona jako **virtual**.
- Metoda w klasie pochodnej powinna mieć identyczną sygnaturę (nazwa, parametry, typ zwracany).
- Opcjonalnie użyj override w C++11+, aby kompilator sprawdził poprawność.

**Nadpisywanie umożliwia polimorfizm: obiekt klasy pochodnej może być traktowany jak obiekt bazowy, ale wywoła nadpisaną metodę.**

#### Przykład:

```
#include <iostream>  
  
class Kształt {  
public:  
    virtual void rysuj() {  
        std::cout << "Rysuje kształt (z klasy bazowej)." << std::endl;  
    }  
};  
  
class Kolo : public Kształt {  
public:  
    void rysuj() override {  
        std::cout << "Rysuje kolo (z klasy pochodnej)." << std::endl;  
    }  
};  
  
int main() {  
    Kształt* ptr = new Kolo(); // Polimorfizm  
    ptr->rysuj(); // Wywoła wersję z Kolo: "Rysuje kolo."  
    delete ptr;  
    return 0;  
}
```

Bez virtual wywołałaby się metoda z Kształt (overloading, nie overriding).



**Deklaracja wskaźnika:** `Kształt* ptr` – Tworzy wskaźnik `ptr` typu `Kształt*`. Oznacza to, że `ptr` może wskazywać na obiekt klasy `Kształt` lub dowolnej klasy pochodnej od `Kształt` (dzięki dziedziczeniu).

- **Alokacja pamięci:** `new Kolo()` – Używa operatora `new`, aby dynamicznie alokować pamięć dla nowego obiektu klasy `Kolo`. `Kolo` jest klasą pochodną od `Kształt`, więc obiekt `Kolo` zawiera wszystkie elementy `Kształt` plus swoje własne.
- **Przypisanie:** `= new Kolo()` – Wskaźnik `ptr` (typu `Kształt*`) jest przypisywany do adresu nowo utworzonego obiektu `Kolo`. To jest dozwolone dzięki **upcastingowi** (konwersji w górę hierarchii dziedziczenia) – obiekt pochodny może być traktowany jak obiekt bazowy bez utraty informacji.

W rezultacie:

- `ptr` "widzi" obiekt jako `Kształt`, ale faktycznie pod spodem jest to obiekt `Kolo`.
- Bez polimorfizmu wywołałbyś metody z `Kształt`, ale dzięki niemu – metody nadpisane z `Kolo`.

## Lekcja 11

### Temat: Polimorfizm w C++

**Polimorfizm** to jedno z kluczowych pojęć w programowaniu obiektowym (OOP) — obok **dziedziczenia** i **enkapsulacji**.

W C++ odgrywa ogromną rolę, zwłaszcza przy projektowaniu elastycznego, rozszerzalnego kodu.

Słowo **polimorfizm** pochodzi z greckiego: "*poly*" – wiele, "*morphe*" – formy. Oznacza, że **ten sam interfejs (np. metoda)** może mieć **różne zachowania**, w zależności od typu obiektu, który ją wywołuje.

W C++ mamy **dwa główne rodzaje** polimorfizmu:

Rodzaj	Nazwa	Kiedy działa	Jak działa
♦ <b>Statyczny</b>	<i>Compile-time</i>	w czasie kompilacji	Przeciążanie funkcji, szablony (templates), przeciążanie operatorów
♦ <b>Dynamiczny</b>	<i>Run-time</i>	w czasie działania programu	Funkcje wirtualne i dziedziczenie klas

## 1. Polimorfizm statyczny (kompilacyjny)

Działa **na etapie kompilacji**.

Kompilator sam wybiera, którą wersję funkcji lub operatora wywołać — na podstawie **typu argumentów**.

### ✓ przykład — przeciążanie funkcji

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

void pokaz(int x) {
    cout << "Liczba całkowita: " << x << endl;
}

void pokaz(double x) {
    cout << "Liczba zmiennoprzecinkowa: " << x << endl;
}

int main() {
    pokaz(10);    // wywoła wersję dla int
    pokaz(3.14);  // wywoła wersję dla double
    return 0;
}
```

- ♦ Kompilator sam wybiera odpowiednią wersję — **polimorfizm kompilacyjny**.

### ✓ Przykład — przeciążanie operatorów

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

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

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

int main() {
    Punkt p1(2, 3), p2(4, 1);
    Punkt suma = p1 + p2; // używa przeciążonego operatora +
    cout << "(" << suma.x << ", " << suma.y << ")" << endl;
}
```

♦ Operator + zachowuje się różnie w zależności od kontekstu — to też **forma polimorfizmu**.

## ✖ 2. Polimorfizm dynamiczny (uruchomieniowy)

Ten rodzaj polimorfizmu działa **w czasie wykonywania programu**.  
Opiera się na **dziedziczeniu** i **funkcjach wirtualnych**.

Dzięki temu możemy wywoływać metody klas pochodnych przez **wskaźnik lub referencję** do klasy bazowej.

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

class Zwierze {
public:
    virtual void dajGlos() {
        cout << "Zwierze wydaje dźwięk." << endl;
    }
};


class Pies : public Zwierze {
public:
    void dajGlos() override {
        cout << "Hau hau!" << endl;
    }
};

class Kot : public Zwierze {
public:
    void dajGlos() override {
        cout << "Miau!" << endl;
    }
};

void wydajDzwiek(Zwierze* z) {
    z->dajGlos();
}

int main() {
    Pies p;
    Kot k;

    wydajDzwiek(&p); // Hau hau!
    wydajDzwiek(&k); // Miau!
}
```

 Tutaj:

- `wydajDzwiek()` nie wie, czy ma do czynienia z `Pies` czy `Kot`.
- Dzięki **funkcjom wirtualnym**, wywoływana jest **odpowiednia implementacja** w czasie działania programu.

## Zastosowanie polimorfizmu

### 1. Tworzenie elastycznych i rozszerzalnych systemów

- Możesz pisać kod działający na poziomie klasy bazowej, który obsługuje różne typy pochodne.
- Przykład: Zwierze → Pies, Kot, Ptak.

### 2. Projektowanie interfejsów i klas abstrakcyjnych

- Klasy z metodami czysto wirtualnymi (= 0) pozwalają definiować wspólne zachowanie bez implementacji.

### 3. Programowanie zorientowane na interfejsy (ang. interface-based programming)

- Dzięki polimorfizmowi możesz łatwo wymieniać moduły, bez zmiany reszty kodu.

### 4. Wzorce projektowe

- Polimorfizm jest podstawą wzorców takich jak *Factory*, *Strategy*, *Observer*, *Command* itp.

## Podsumowanie

Typ polimorfizmu	Kiedy działa	Mechanizm	Przykład
<b>Statyczny (kompilacyjny)</b>	W czasie kompilacji	Przeciążanie funkcji, operatorów, szablony	<code>void pokaz(int); void pokaz(double);</code>
<b>Dynamiczny (uruchomieniowy)</b>	W czasie działania programu	Funkcje wirtualne, dziedziczenie	<code>virtual void dajGlos();</code>

