

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.