

PROGRAMOWANIE I

Podstawy programowania w C++

Marek Brancewicz

5 listopada 2026

Wydział Fizyki
Uniwersytet w Białymstoku

Wykład 3

Plan wykładu

1. Pętle: `for`, `while`, `do ... while`.
2. Przerywanie pętli: `break`, `continue`. Pętle zagnieżdżone.
3. Liczby pseudolosowe: `rand()`, `srand()`.
4. Instrukcja wielokrotnego wyboru `switch ... case`.
5. Pomiar czasu wykonywania programu.

Pętle

Po co są pętle?

Pętla instrukcja, która powtarza wykonanie bloku kodu, dopóki spełniony jest warunek

Iteracja jednokrotne wykonanie bloku pętli

Zmienna sterująca (licznik) zmienna, od której zależy, czy pętla wykona kolejną iterację, np. `i`

Pętla	**Warunek sprawdzany**	**Typowe zastosowanie**
`for`	przed iteracją	znana liczba powtórzeń
`while`	przed iteracją	nieznana liczba powtórzeń
`do ... while`	po iteracji	co najmniej jedno wykonanie

Pętla `for`

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

`int i = 1`	inicjalizacja – wykonywana **raz**, przed pętlą
`i <= 5`	warunek – sprawdzany **przed** każdą iteracją; `false` kończy pętlę
`i++`	krok – wykonywany **po** każdej iteracji

Kolejność: inicjalizacja → warunek → blok → krok → warunek → blok → ...

Zmienna `i` zadeklarowana w `for` istnieje tylko wewnątrz pętli.

Pętla `for` – przykłady

```
for (int i = 10; i >= 0; i -= 2)      // od 10 w dół, co 2
    cout << i << " ";                 // 10 8 6 4 2 0

int n = 100, suma = 0;
for (int i = 1; i <= n; i++)          // suma liczb od 1 do n
    suma += i;
cout << suma << endl;                 // 5050
```

`i++`	inkrementacja	to samo co `i = i + 1` lub `i += 1`
`i--`	dekrementacja	to samo co `i = i - 1` lub `i -= 1`

Zmienna sterująca nie musi być typu `int` ani zmieniać się o 1 – krok może być dowolną instrukcją.

Pętla `while`

```
int liczba, suma = 0;
cout << "Podaj liczbe (0 = koniec): ";
cin >> liczba;
while (liczba != 0)
{
    suma += liczba;
    cout << "Podaj liczbe (0 = koniec): ";
    cin >> liczba;
}
cout << "Suma: " << suma << endl;
```

Warunek jest sprawdzany **przed** każdą iteracją – jeśli od razu jest fałszywy, blok nie wykona się ani razu (tu: gdy pierwszą liczbą jest 0). Liczba powtórzeń zależy od danych, a nie od licznika.

Pętla `do ... while`

```
char odp;
do
{
    cout << "Hello, World!" << endl;
    cout << "Jeszcze raz? (t/n): ";
    cin >> odp;
} while (odp == 't' || odp == 'T');      // średnik!
```

Warunek jest sprawdzany **po** iteracji, więc blok wykona się **co najmniej raz**. To naturalny wybór, gdy warunek zależy od czegoś, co dopiero powstaje w bloku (tu: odpowiedź użytkownika).

Po `while (...)` w pętli `do ... while` **musi** być średnik.

Pętle

`while` a `do ... while`

Ten sam warunek, fałszywy od początku:

```
int i = 10;
while (i < 5)
{
    cout << "while";
    i++;
}
// nie wypisze nic
```

```
int j = 10;
do
{
    cout << "do-while";
    j++;
} while (j < 5);
// wypisze: do-while
```

W obu pętlach blok musi zmieniać coś, od czego zależy warunek (tu: `i++`, `j++`) – inaczej pętla nigdy się nie skończy.

Przerywanie pętli: `break` i `continue`

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

```
for (int i = 1; i <= 6; i++)
{
    if (i == 3) continue;
    cout << i << " ";
}
// 1 2 4 5 6
```

`break` – natychmiast kończy pętlę; program przechodzi do instrukcji za pętlą.

`continue` – pomija resztę bieżącej iteracji i przechodzi do następnej (w `for` wykonuje się jeszcze krok `i++`).

W pętlach zagnieżdżonych oba działają tylko na pętlę, w której się znajdują.

Pętla nieskończona

```
int x;
while (true)                        // lub: for (;;)
{
    cout << "Podaj liczbe dodatnia: ";
    cin >> x;
    if (x > 0) break;               // jedyne wyjście z pętli
    cout << "To nie jest liczba dodatnia!" << endl;
}
```

Taka pętla jest poprawna, gdy warunek wyjścia wygodniej sprawdzić w środku bloku.

Uwaga: po wpisaniu liter `cin` wpada w stan błędu i ta pętla kręci się bez końca (pomoże `cin.fail()`). Zawieszony program przerywa **Ctrl+C**.

Pętle zagnieżdżone (pętla w pętli)

```
for (int i = 1; i <= 4; i++)          // wiersze
{
    for (int j = 1; j <= 5; j++)      // kolumny
    {
        cout << i * j << "\t";        // \t = tabulator
    }
    cout << endl;                     // koniec wiersza
}
```

```
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
```

Dla każdej iteracji pętli zewnętrznej pętla wewnętrzna wykonuje się **w całości**: blok wewnętrzny wykona się $4 \cdot 5 = 20$ razy.

Typowe błędy w pętlach

Średnik po nagłówku pętli – pętla steruje pustą instrukcją:

```
for (int i = 1; i <= 5; i++);    // blok niżej wykona się raz
while (i < 5);                   // nieskończona, bez ostrzeżenia!
```

Pomyłka o jeden – `i < 10` czy `i <= 10`? Start od 0 czy od 1? Sprawdź pierwszą i ostatnią iterację.

Porównywanie liczb rzeczywistych:

```
for (double x = 0.0; x != 1.0; x += 0.1)  // nie kończy się
```

$10 \times 0{,}1$ daje w komputerze `0.9999999999999989`, a nie `1.0`. Liczby `double` porównujemy przez `<` lub `<=`, a pętlę najlepiej sterować licznikiem `int`.

Liczby pseudolosowe

Czy komputer może losować?

Komputer wykonuje instrukcje deterministycznie – ten sam program z tymi samymi danymi daje ten sam wynik. Dlatego zamiast liczb losowych używa się **pseudolosowych**:

Generator algorytm, który z poprzedniej liczby oblicza następną; kolejne liczby *wyglądają* na losowe

Ziarno (ang. *seed*) liczba, od której generator zaczyna; to samo ziarno daje zawsze ten sam ciąg

Okres po pewnej liczbie kroków ciąg zaczyna się powtarzać

Żeby przy każdym uruchomieniu programu ciąg był inny, ziarnem jest zwykle aktualny czas: *Unix time* – liczba sekund od 1.01.1970 (5.11.2026 o północy UTC: 1 793 836 800).

Przykład: generator liniowy kongruencyjny

$$x_{k+1} = (a \cdot x_k + c) \bmod m$$

x_0 – ziarno, a, c, m – stałe, mod – reszta z dzielenia (w C++: %).
Wyniki są z zakresu $0 \ldots m-1$.

Przykład dla $a = 5$, $c = 3$, $m = 16$, $x_0 = 7$:

$x_1 = (5 \cdot 7 + 3) \bmod 16 = 38 \bmod 16 = 6$
$x_2 = (5 \cdot 6 + 3) \bmod 16 = 33 \bmod 16 = 1$
$x_3 = (5 \cdot 1 + 3) \bmod 16 = 8, \ldots$

Cały ciąg: `6 1 8 11 10 5 12 15 14 9 0 3 2 13 4 7`, potem znów `6 1 8 ...` (okres 16). Prawdziwe generatory mają duże stałe (np. $m = 2^{31}$) i bardzo długi okres.

Liczby pseudolosowe

`rand()` i `srand()`

```
#include <cstdlib>          // rand(), srand(), RAND_MAX
#include <ctime>            // time()
...
srand(time(nullptr));       // ziarno = czas; RAZ w programie
for (int i = 1; i <= 10; i++)
    cout << rand() % 6 + 1 << " ";      // rzut kostką: 1..6
```

`rand()`	kolejna liczba pseudolosowa z zakresu 0 . . . `RAND_MAX`
`srand(z)`	ustawia ziarno z
`time(nullptr)`	aktualny czas w sekundach (Unix time)
`RAND_MAX`	Windows (MinGW, MSVC): **32 767**
	Linux (GCC): 2 147 483 647

Losowanie liczby z zakresu

```
rand() % n                    // od 0 do n-1
rand() % n + 1                // od 1 do n
rand() % (b - a + 1) + a      // od a do b

// przykład: od 51 do 75 (75 - 51 + 1 = 25 liczb)
rand() % 25 + 51              // 0..24  ->  51..75
```

- `rand() % n` to reszta z dzielenia przez `n`, czyli $0 \ldots n-1$; dodanie a przesuwa zakres.
- % ma wyższy priorytet niż +: `rand() % n + 1` to `(rand() % n) + 1`.
- W Windows `rand()` zwraca najwyżej 32 767 – tak nie wylosujemy np. liczby z zakresu $0 \ldots 99\,999$.

Typowe błędy przy losowaniu

Brak `srand` – przy każdym uruchomieniu ten sam ciąg:

```
cout << rand() % 6 + 1;      // za każdym razem: 2 5 4 2 6 2 ...
```

`srand` **w pętli** – w ciągu tej samej sekundy ziarno jest takie samo, więc `rand()` zwraca wciąż tę samą liczbę:

```
for (int i = 1; i <= 10; i++)
{
    srand(time(nullptr));              // BŁĄD!
    cout << rand() % 6 + 1 << " ";  // 6 6 6 6 6 6 6 6 6 6
}
```

Zasada: `srand(time(nullptr));` wywołujemy **raz**, na początku `main()`. (Brak `srand` bywa zaletą: powtarzalny ciąg ułatwia szukanie błędów.)

Dla dociekliwych: biblioteka <random>

```
#include <random>
...
random_device rd;                          // źródło ziarna
mt19937 gen(rd());                         // generator
uniform_int_distribution<int> k(1, 6);  // liczby 1..6
for (int i = 1; i <= 10; i++)
    cout << k(gen) << " ";
```

Nowoczesny sposób (od C++11): lepsza jakość liczb, okres $2^{19937} - 1$, dowolny zakres bez % i bez ograniczenia RAND_MAX.

Na zajęciach wystarczy rand().

Instrukcja switch

Instrukcja wielokrotnego wyboru `switch`

```
switch (dzien)                  // int dzien; - wyrażenie sterujące
{
    case 1:                     // gdy dzien == 1
        cout << "Poniedzialek" << endl;
        break;                  // wyjście ze switch
    case 6:                     // kilka wartości -> ten sam kod
    case 7:
        cout << "Weekend!" << endl;
        break;
    default:                    // gdy żaden case nie pasuje
        cout << "Inny dzien lub bledny numer" << endl;
}
```

Jak działa `switch`? Rola `break`

```
int x = 1;
switch (x)
{
    case 1:
        cout << "jeden" << endl;
        // brak break!
    case 2:
        cout << "dwa" << endl;
        break;
}
```

Wynik: `jeden`, `dwa`.

`switch` skacze do pasującego `case`, a potem wykonuje kolejne instrukcje **aż do** `break` lub końca bloku – także te z następnych `case` (ang. *fall-through*).

Brak `break` to częsty błąd (`-Wextra` ostrzega: `may fall through`).

Celowo – przy pustych `case`: `case 6: case 7:`.

Ograniczenia `switch`

- Wyrażenie sterujące musi być typu całkowitego (`int`, `char`, `long`, ...) – **nie** `double`, **nie** `string`.
- Przy `case` – stałe (`case 1:`, `case 'a':`), bez powtórzeń.
- Tylko równość – bez zakresów (`case x > 10:` to błąd).

Typ `char` pozwala łatwo obsłużyć duże i małe litery:

```
switch (odp)                    // char odp;
{
    case 't': case 'T': cout << "Tak" << endl; break;
    case 'n': case 'N': cout << "Nie" << endl; break;
    default:  cout << "Nieznana odpowiedz" << endl;
}
```

Instrukcja switch

`switch` **czy** `if ... else if`?

`switch`	jedna zmienna porównywana z listą konkretnych wartości, np. wybór opcji z menu, numer dnia, litera
`if ... else if`	zakresy (`punkty >= 90`), warunki złożone (`&&`, `||`), liczby rzeczywiste, napisy

Każdy `switch` da się zapisać jako ciąg `if ... else if`, ale nie odwrotnie. `switch` jest czytelniejszy, gdy przypadków jest dużo.

Wszystko razem: menu w pętli

```
int wybor;                          // wcześniej: srand(time(nullptr));
do
{
    cout << "1 - kostka, 2 - liczba 1..100, 0 - koniec: ";
    cin >> wybor;
    switch (wybor)
    {
        case 1:  cout << rand() % 6 + 1 << endl;   break;
        case 2:  cout << rand() % 100 + 1 << endl; break;
        case 0:  cout << "Koniec." << endl;        break;
        default: cout << "Nie ma takiej opcji!" << endl;
    }
} while (wybor != 0);
```

`case 0` jest potrzebny, żeby wybór „koniec” nie trafił do `default`.

Pomiar czasu

Pomiar czasu wykonywania: `clock()`

```
#include <ctime>                    // clock(), clock_t, CLOCKS_PER_SEC
...
double suma = 0.0;
clock_t start = clock();                              // początek pomiaru
for (int i = 1; i <= 100000000; i++)
    suma += 1.0 / i;
clock_t stop = clock();                               // koniec pomiaru
double czas = double(stop - start) / CLOCKS_PER_SEC;
cout << "Suma: " << suma << endl;
cout << "Czas: " << czas << " s" << endl;
```

`clock()`	liczba „tyknięć zegara” od startu programu
`CLOCKS_PER_SEC`	liczba tyknięć na sekundę
`double(...)`	konieczne – inaczej dzielenie byłoby całkowite

Pomiar czasu – uwagi

- W Windows `clock()` mierzy czas rzeczywisty, w Linuksie – czas pracy procesora (bez czekania, np. na klawiaturę).
- Wynik zależy od komputera i od opcji kompilacji: ten sam program skompilowany z `-O2` (optymalizacja) może działać kilka razy szybciej (w teście: 0,29 s $\rightarrow$ 0,13 s).
- Kompilator z optymalizacją może usunąć pętlę, której wynik nie jest używany – dlatego w przykładzie wypisujemy `suma`.
- Krótkie operacje (ułamek milisekundy) mierzymy, powtarzając je wiele razy w pętli i dzieląc czas przez liczbę powtórzeń.
- Dokładniejszy pomiar czasu rzeczywistego: biblioteka `<chrono>` (`steady_clock`).

Przydatne funkcje (do zadań)

`exit(0);`	`<cstdlib>`	kończy program z kodem 0
`system("cls");`	`<cstdlib>`	czyści ekran (Windows)
`system("clear");`	`<cstdlib>`	czyści ekran (Linux)
`Sleep(1000);`	`<windows.h>`	czeka 1000 ms
`Beep(2000, 500);`	`<windows.h>`	2000 Hz przez 500 ms
`_getch();`	`<conio.h>`	czeka na klawisz (bez Entera)

`<windows.h>` i `<conio.h>` nie należą do standardu C++ – program z nimi nie skompiluje się w Linuksie. Przenośne czekanie:

```
#include <thread>
#include <chrono>
this_thread::sleep_for(chrono::milliseconds(1000));
```

Podsumowanie

- `for` – znana liczba powtórzeń; `while` – warunek przed iteracją (może nie wykonać się wcale); `do ... while` – co najmniej raz.
- `break` kończy pętlę, `continue` przechodzi do następnej iteracji.
- Komputer generuje liczby *pseudolosowe* – ten sam ciąg dla tego samego ziarna.
- `srand(time(nullptr))` raz na początku, `rand() % n + a` – liczba z zakresu.
- `switch` wybiera kod według wartości całkowitej; bez `break` wykonują się kolejne `case`.
- Czas wykonania mierzymy funkcją `clock()`.

Pytania kontrolne

1. Ile razy wykona się pętla `for (int i = 0; i < 10; i += 3)`?
2. Czym różni się `while` od `do ... while`? Kiedy wybrać którą?
3. Co robi `break`, a co `continue`?
4. Dlaczego `rand()` daje liczby *pseudolosowe*? Co to jest ziarno?
5. Dlaczego `srand()` nie wywołuje się w pętli?
6. Jak wylosować liczbę całkowitą z zakresu $-5 \ldots 5$?
7. Co się stanie, gdy w `case` zabraknie `break`?
8. Dlaczego `switch` nie działa ze zmienną typu `double`?

Zadania

Zadanie 4: odliczanie do startu rakiety

Napisz program, który odlicza od 10 do 1 co 1 sekundę (z dźwiękiem) i wypisuje `START`, gdy licznik dojdzie do 0. Przed każdą liczbą wyczyść ekran.

Przydatne (Windows):

```
#include <windows.h>
Sleep(1000);            czeka 1000 ms
Beep(2000, 500);        dźwięk 2000 Hz przez 500 ms
system("cls");          czyści ekran
```

Zadanie 5: populacja bakterii

Bakteria dzieli się na dwie co godzinę. Napisz program, który wypisuje liczebność populacji po kolejnych godzinach, aż przekroczy ona 1 000 000 000 organizmów. Zastosuj pętlę `while`.

```
0 h - populacja = 1
1 h - populacja = 2
2 h - populacja = 4
3 h - populacja = 8
...
```

Jak zmieni się program, jeśli granicą będzie 10 000 000 000? Sprawdź, co się stanie dla typu `int`, i porównaj z `long long`.

Zadanie 6: sprawdzanie danych w pętli

a) Program prosi o liczbę dodatnią. Jeśli użytkownik poda liczbę ujemną lub zero, prosi ponownie – aż do podania poprawnej liczby albo do przekroczenia ustalonej liczby nieudanych prób. Zastosuj pętlę `do ... while`.

b) Program prosi o liczbę rzeczywistą. Jeśli użytkownik wpisze coś, co nie jest liczbą (np. litery), program prosi ponownie – aż do podania poprawnej liczby albo do przekroczenia ustalonej liczby prób. Wykorzystaj `cin.fail()`, `cin.clear()` i `cin.ignore()`.

Zadanie 7: LOTTO

Napisz program, który losuje 6 liczb spośród 49 i wypisuje je na ekranie. Czy program na pewno działa jak losowanie LOTTO? Jeśli nie – dlaczego? Popraw go.

Zadanie 8: zgadywanie liczby

Program losuje liczbę od 1 do 100 i prosi użytkownika o jej odgadnięcie. Po każdej próbie wypisuje „Za duża", „Za mała" albo „Zgadłeś!". Po odgadnięciu wypisuje, w której próbie się udało. Zastosuj pętlę `while` lub `do ... while` i instrukcję `if`.

Ile prób wystarczy w najgorszym przypadku przy najlepszej strategii?

Zadanie 9: losowe liczby rzeczywiste

Napisz program, który losuje liczby rzeczywiste z zakresu podanego przez użytkownika. Wykorzystaj losowanie liczb całkowitych poznane na wykładzie i odpowiednie działania matematyczne. Ile różnych wartości może dać Twój program w Windows?

Zadanie 10: kalkulator

Napisz kalkulator z menu dostępnych działań, wykorzystując instrukcję `switch`. Program działa w pętli, dopóki użytkownik nie wybierze opcji WYJŚCIE. Zadbaj o dzielenie przez zero.

Zadanie 11: liczba π metodą Monte Carlo

1. Wylosuj n punktów (x, y) w kwadracie $[-1, 1] \times [-1, 1]$.
2. Policz, ile z nich (m) leży w kole o promieniu 1 i środku $(0, 0)$, czyli spełnia $x^2 + y^2 \leq 1$.
3. Stosunek pól koła i kwadratu wynosi $\pi/4$, więc $\pi \approx 4m/n$.

Program ma liczyć π w pętli dla $n = 10, 100, 1000, \ldots$ (kolejne potęgi 10), tak aby całość trwała około minuty. Dla każdego n wypisz: numer iteracji, n, obliczone π, błąd względny w procentach i czas obliczeń.

Wskazówki: jakiego typu musi być n dla $n = 10^{10}$? Jak dokładnie da się przybliżyć π, skoro w Windows `rand()` daje tylko 32 768 różnych wartości?