
PROGRAMOWANIE
1
Marek Brancewicz
m.brancewicz@uwb.edu.pl
WYDZIAŁ FIZYKI
UNIWERSYTET W BIAŁYMSTOKU
https://pl.freepik.com

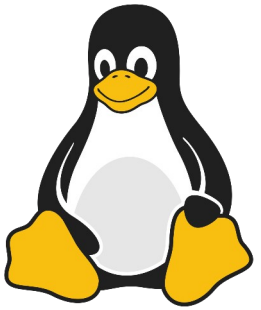
WYKŁAD

1

(01.10.2025)

WPROWADZENIE

1. Organizacja zajęć, zasady oceniania
2. Literatura, kursy on-line, zasady korzystania z AI
3. Podstawowe pojęcia
4. Oprogramowanie (MinGw, CodeBlocks)
5. Praca w terminalu (Linux)



MINGW-W64

CODE BLOCKS

HARMONOGRAM WYKŁADÓW

Wykład	Data	Tematy
1	01.10.2025	Wprowadzenie. Podstawowe pojęcia. Oprogramowanie. Praca w terminalu Linux'a (lub wierszu poleceń Windows). Pierwszy program. Podstawowe typy zmiennych. Wczytywanie danych z klawiatury i wypisywanie ich na ekranie (strumień wejścia-wyjścia). Instrukcja warunkowa „if".
2	15.10.2025	Pętle (for, while, do … while). Liczby pseudolosowe. Instrukcja wielokrotnego wyboru „switch-case". Pomiar czasu wykonywania programu.
3	29.10.2025	Tablice. Zapisywanie do i czytanie z pliku tekstowego. Złożone typy danych 1 (string).
4	26.11.2025	Funkcje (definicja, struktura, przykłady, przekazywanie tablic do funkcji, wartości domyślne parametrów funkcji, przeciążanie funkcji, sufiksy danych, przesłanianie zmiennych). Rekurencja (rekursja).
5	10.12.2025	Wskaźniki.Dynamiczna alokacja pamięci.
6	21.01.2026	
7	04.02.2026	

LIERATURA (PL & ENG)

[1] kursy on-line

[2] modele językowe AI

[3] W. Porębski, *Język C++ : wprowadzenie do programowania*, wyd. 2, Komputerowa Oficyna Wydawnicza "Help", Warszawa 1999

[4] J. Grębosz, *Symfonia C ++ standard : programowanie w języku C++ orientowane obiektowo*, Wydawnictwo "Edition 2000" : Oficyna Kallimach, Kraków 2005

[5] S. Prata, *Język C++*, wyd. 5, Wydawnictwo Helion, Gliwice 2006

[6] A. Koenig, *Accelerated C++ : practical programming by example*, 22nd printing, Addison-Wesley, Boston 2013

PODSTAWOWE POJĘCIA

Język programowania Zbiór zasad określających, kiedy ciąg symboli tworzy **program komputerowy** oraz jakie obliczenia opisuje.

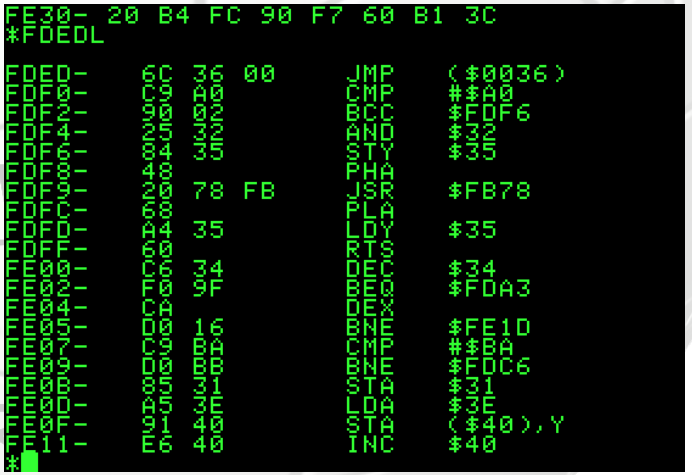
Program komputerowy Sekwencja symboli opisująca realizowanie obliczeń zgodnie z pewnymi regułami zwanymi **językiem programowania**. Program jest zazwyczaj wykonywany przez komputer (np. wyświetlenie strony internetowej), zwykle bezpośrednio, jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest interpretowany przez inny program (**interpreter**). Program może być ciągiem instrukcji opisujących modyfikacje stanu maszyny, ale może również opisywać obliczenia w inny sposób (np. rachunek lambda)

Kod źródłowy Zapis programu komputerowego przy pomocy określonego **języka programowania**, opisujący operacje, jakie powinien wykonać komputer na zgromadzonych lub otrzymanych danych. Kod źródłowy jest wynikiem pracy programisty i pozwala wyrazić w czytelnej dla człowieka formie strukturę oraz działanie **programu komputerowego**. Jest on zwykle zapisywany w pliku tekstowym.

Kompilator Program służący do automatycznego tłumaczenia kodu napisanego w jednym języku (**języku źródłowym**) na równoważny kod w innym języku (**języku wynikowym**). Proces ten nazywany jest kompilacją. W informatyce kompilatorem nazywa się najczęściej program do tłumaczenia **kodu źródłowego** w **języku programowania** na **język maszynowy**. Niektóre z nich tłumaczą najpierw do **języka asemblera**, a ten na język maszynowy jest tłumaczony przez asembler.

Język maszynowy (kod maszynowy)

Zestaw rozkazów procesora, w którym zapis programu wyrażony jest w postaci liczb binarnych stanowiących rozkazy oraz ich argumenty.



Zintegrowane środowisko programistyczne

(**IDE**, od ang. integrated development environment) – program lub zespół programów (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania.

Interpreter Analizuje kod źródłowy programu, a przeanalizowane fragmenty wykonuje. Realizowane jest to w inny sposób niż w procesie **kompilacji**, podczas którego nie wykonuje się wejściowego programu (**kodu źródłowego**), lecz tłumaczy go do wykonywalnego **kodu maszynowego** lub kodu pośredniego, który jest następnie zapisywany do pliku w celu późniejszego wykonania.

Wykonanie programu za pomocą interpretera jest wolniejsze, a do tego zajmuje więcej zasobów systemowych niż wykonanie kodu skompilowanego, lecz może zająć relatywnie mniej czasu niż **kompilacja** i uruchomienie. Jest to zwłaszcza ważne przy tworzeniu i testowaniu kodu, kiedy cykl edycja-interpretacja-**debugowanie** może często być znacznie krótszy niż cykl edycja-kompilacja-uruchomienie-**debugowanie**.

Debugger Program komputerowy służący do dynamicznej analizy innych programów, w celu odnalezienia i identyfikacji zawartych w nich błędów, zwanych z angielskiego bugami (robakami). Proces nadzorowania wykonania programu za pomocą debuggera określa się mianem **debugowania**. Debugger jest standardowym wyposażeniem większości współczesnych **środowisk programistycznych**.

OPROGRAMOWANIE

MinGW (Minimalist GNU for Windows) – to środowisko programistyczne umożliwiające kompilację kodu źródłowego w systemie Windows za pomocą narzędzi GNU, takich jak GCC (GNU Compiler Collection), GDB czy Make. Dostarcza minimalny zestaw bibliotek i nagłówków umożliwiających tworzenie natywnych aplikacji Windows (bez potrzeby użycia warstwy zgodności, jak Cygwin). MinGW obsługuje języki C, C++, Fortran i inne, a jego rozszerzeniem jest MinGW-w64, wspierające zarówno architektury 32-, jak i 64-bitowe.

Code::Blocks – to darmowe, wieloplatformowe środowisko programistyczne (IDE) przeznaczone głównie do programowania w C, C++ i Fortranie. Oferuje edytor kodu z podświetlaniem składni, debugger, system projektów oraz integrację z popularnymi kompilatorami, takimi jak GCC (MinGW) czy Clang. Dzięki modułowej budowie (pluginy) można je łatwo rozszerzać o dodatkowe funkcje.

PRACA W TERMINALU (LINUX)

Terminal to tekstowy interfejs użytkownika, który wyświetla tekst, pozwala wpisywać komendy i przekazuje je dalej do powłoki (np. bash).

Bash (Bourne Again SHell) to powłoka systemowa (shell) czyli program pośredniczący między użytkownikiem a jądrem systemu Linux. W skrócie – Bash to program, który interpretuje (CLI - Command Line Interpreter) komendy wpisywane w terminalu.

Bash to nie tylko interpreter ale również pełnoprawny język skryptowy, w którym sekwencję poleceń można zapisać do pliku *.sh, (np. run.sh):

```
#!/bin/bash
echo "Kompiluję program..."
g++ main.cpp -o main
echo "Uruchamiam:"
./main
```

#! – shebang (sharp+bang), mówi systemowi, który interpreter ma wykonać skrypt

A następnie nadać (+) temu skryptowi uprawnienia do wykonywania (x) poleceniem chmod (change mode):

```
chmod +x run.sh
./run.sh
```

Podstawy nawigacji w terminalu Linuxa

```
pwd
ls
cd folder
cd ..
g++ --version
g++ -v
```

- wyświetla katalog bieżący
- wyświetla zawartośc bieżącego katalogu
- przejście do katalogu *folder*
- wyjście z katalogu „piętro wyżej”
- wyświetla wersję kompilatora g++
- szczegółowe informacje o środowisku kompilacji

chamianie programu w terminalu

```
g++ program.cpp
```

- kompilacja domyślna, tworzy plik wykonywalny o nazwie *a*

```
g++ program.cpp -o program
```

- kompilacja z tworzeniem pliku wyjściowego o nazwie *program*

Jeżeli polecenia będą w skrypcie basha, np. *compile.sh*:

```
chmod +x compile.sh
./ compile.sh
```

- nadaj skryptowi atrybut wykonywalności
- uruchom skrypt

PRACA W WIERSZU POLECEŃ (cmd) lub PowerShell (WINDOWS)

CMD (Command Prompt) to najstarszy i najprostszy odpowiednik terminala i powłoki w Windows, gdzie skryptami są pliki tekstowe *.bat lub *.cmd. Prosty, dostępny ale o dość ograniczonej funkcjonalności i niekompatybilny z systemami opartymi na poleceniach Unix-a (pierwowzór Linux-a, Linux jest jego wolną implementacją).

PowerShell to nowoczesny odpowiednik Bash-a stworzony przez Microsoft. Terminal: PowerShell Console (albo Windows Terminal), powłoka: powershell.exe (lub pwsh), skrypty: *.ps1. Jest potężnym językiem skryptowym z obsługą obiektów (nie tylko tekst jak w Bashu). Nie jest kompatybilny z poleceniami Unix-a i ma mniej zwięzłą składnię.

Istnieją tzw. emulatory Basha dla Windows, np. **Git Bash** (niemal pełna zgodność z Bash) lub **WSL** (Windows Subsystem for Linux), który zapewnia 100% zgodności z Bash-em.

PowerShell Execution Policy

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Trwałe włączenie wykonywania skryptów PowerShell dla bieżącego użytkownika (najczęściej używane przez programistów).

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

Włączenie wykonywania skryptów PowerShell w bieżącej sesji (po zamknięciu okna wraca do domyślnych ustawień).

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Restricted
```

Wyłączenie wykonywania skryptów PowerShell dla bieżącego użytkownika (powrót do domyślnych ustawień).

Przewaga terminala nad IDE

1) **Pełna kontrola nad kompilacją i linkowaniem.** Programista widzi wszystkie opcje, sam je wybiera i decyduje o wszystkim, ma pełna kontrolę nad optymalizacją, co jest kluczowe dla wydajności i przenośności kodu, pomaga zrozumieć strukturę projektu, etapy kompilacji i błędy. IDE ukrywa ten proces co jest wygodne ale mniej edukacyjne.

2) **Automatyzacja i powtarzalność** (Makefile, Bash). W terminalu łatwo tworzy się skrypty, które automatyzują budowanie i testowanie aplikacji. Terminalowe skrypty są lekkie i przenośne. IDE trzeba konfigurować ręcznie na każdym komputerze.

3) **Praca na zdalnych systemach** (np. serwery obliczeniowe, HPC). Na klastrach, superkomputerach, serwerach linuksowych nie ma środowisk graficznych (GUI). Tam jedynym sposobem jest terminal (SSH + kompilacja wierszem poleceń). Dlatego naukowcy, inżynierowie i administratorzy systemów muszą umieć pracować w CLI (np.. Bash). Nikt na klastrze obliczeniowym nie uruchamia IDE.

4) **Terminal jest szybszy i lżejszy**. Nie wymaga graficznego interfejsu, uruchamia się natychmiast, nie zużywa zasobów systemowych. Dla małych testów lub krótkich programów jest po prostu najefektywniejszy.

5) **Uniwersalność i niezależność od środowiska**. Skrypty kompilujące działają na systemach tam, gdzie często nie ma IDE.

PIERWSZY PROGRAM

1. Pierwszy program w C++ (`"Hello world"`)
2. Struktura programu
3. Typy zmiennych, zmienne globalne i lokalne
4. Wczytywanie danych z klawiatury i wypisywanie ich na ekranie.

```
#include <iostream>       // dyrektywy preprocesora

using namespace std;      // przestrzeń nazw std

                    // definicje zmiennych globalnych

int main()
{
     cout << "HELLO WORLD!" << endl;
     return 0;                              // kod błędu
}
```

Struktura programu

- dyrektywy preprocesora (#), np: dołączanie bibliotek do programu (`#include <library>`), definicje stałych (`#define grav 9.81`)
- `using namespace std;` (przestrzeń nazw)
- definicje zmiennych (globalnych) i stałych (np. `int x,n;` `const float pi=3.1415926;`)
- funkcja główna (sterująca) – `int main() {}`
- `return 0;` (zwraca kody błędów)
- białe znaki (spacje, entery) są ignorowane
- każda linia powinna się kończyć średnikiem (`;`) (są wyjątki)
- komentarze: `//` (jedna linia) lub `/*` (początek) i `*/` (koniec)

PODSTAWOWE TYPY ZMIENNYCH

Nazwa	Rozmiar [B] (32 bit)	Rozmiar [B] (64 bit)	Zakres
bool	1	1	prawda (1) albo fałsz (0)
char	1	1	-128 –> 127
unsigned char	1	1	0 –> 255
short	2	2	-32 768 –> 32 767
unsigned short	2	2	0 –> 65 535
int	4	4	-2 147 483 648 –> 2 147 483 647
unsigned int	4	4	0 –> 4 294 967 295
long	4	4	-2 147 483 648 –> 2 147 483 647
unsigned long	4	4	0 –> 4 294 967 295
long long	8	8	-9 223 372 036 854 775 808 –> 9 223 372 036 854 775 807
unsigned long long	8	8	0 –> 18 446 744 073 709 551 615
float	4	4	3.4 E+/-38 (7 cyfr)
double	8	8	1.7 E+/-308 (15 cyfr)
long double	8	12	1.7 E+/-308 (15/20 cyfr)
string	4	4	tablica (łańcuch) znaków (char) [wskaźnik !]

Sprawdzanie rozmiarów zmiennych

```
#include <iostream>
using namespace std;
int main()
{
    cout<<"size of bool = "<<sizeof(bool)<<endl;
    cout<<"size of char = "<<sizeof(char)<<endl;
    cout<<"size of int = "<<sizeof(int)<<endl;
    cout<<"size of long = "<<sizeof(long)<<endl;
    cout<<"size of long long = "<<sizeof(long long)<<endl;
    cout<<"size of float = "<<sizeof(float)<<endl;
    cout<<"size of double = "<<sizeof(double)<<endl;
    cout<<"size of string = "<<sizeof(string)<<endl;
    return 0;
}
```

Zmienna globalna Zmienna istniejąca przez cały czas życia programu i widziana z wielu miejsc w programie. W C++ każda zmienna globalna jest **zmienną statyczną.**

Zmienna lokalna Zmienna zdefiniowana i dostępna wyłącznie w określonym bloku programu, tworzona w momencie wejścia do tego bloku oraz usuwana z pamięci w momencie wyjścia z danego bloku. Tym samym zasięg zmiennej lokalnej oraz czas jej życia pokrywają się i obejmują blok, w którym zmienna lokalna jest zdefiniowana. Zmienna lokalna ma więc określony, ograniczony zakres istnienia i dostępności.

To w jakich blokach programowych można tworzyć zmienne lokalne definiuje składnia konkretnego języka programowania. Typowymi blokami, w których można w różnych językach programowania tworzyć zmienne lokalne, są moduły, podprogramy oraz w pewnych językach programowania także instrukcje blokowe (lub inne instrukcje strukturalne, np. pętla for w języku C.

Zmienna lokalna w danym bloku **przesłania** zdefiniowaną zmienną globalną lub zmienną lokalną z bloku nadrzędnego o tym samym identyfikatorze. Tym samym programista nie może wprost, za pomocą danego identyfikatora, w bloku o zdefiniowanej zmiennej lokalnej, odwołać się do zmiennej zewnętrznej o tym samym identyfikatorze co zdefiniowana zmienna lokalna, choć może to zrobić za pomocą innych konstrukcji, jeżeli są dostępne w danym języku programowania, np. selekcja, wskaźnik, przemianowanie, nakładanie zmiennych lub inne.

WCZYTYWANIE DANYCH
Z KLAWIATURY I WYPISYWANIE ICH NA EKRANIE

```
int ndiv;           // deklaracja zmiennej całkowitej
float x,y,z;        // deklaracja zmiennej zmiennoprzecinkowej
string t;           // deklaracja zmiennej tekstowej (?)
int main()
{
    cout<<"Podaj liczbe : ";
    cin>>x;
    cout<<"Podana liczba to : "<<x<<endl<<endl;
    cout<<"Napisz tekst : ";
    cin>>t;
    cout<<"Napisales : "<<t<<endl;
    return 0;
}
```

INSTRUKCJA WARUNKOWA „if”

Czyli jak nauczyć komputer podejmowania decyzji...

Przykłady:

1. Weryfikacja kodu dostępu (PIN)
2. Logowanie z nazwą użytkownika i hasłem

Typy zmiennych do wykorzystania:

`int` – liczba całkowita (np. `int PIN, pin, Pin;`)

`string` – tekstowa (np. `string password=”a12b34c”;`)

`bool` – logiczna, prawda (1) lub fałsz (0) (np. `bool c;`)

Operatory porównania (wyrażenia boolowskie):

`==`	logiczne porównanie	`(a==b)`
`<`	mniejsze niż	`(a<b)`
`>`	większe niż	`(a>b)`
`<=`	mniejsze lub równe	`(a<=b)`
`>=`	większe lub równe	`(a>=b)`
`!=`	nie równe	`(a!=b)`

Operatory logiczne:

`&&` - iloczyn logiczny "i"

`||` - suma logiczna "lub"

`!` - zaprzeczenie logiczne (negacja) "nie"

Pytanie:

Jaka zmienna będzie odpowiednia do przechowywania PIN?

Ćwiczenie:

`x=3, a=7`

Określ wartość wyrażenia logicznego:

```
!((x<=5)||(x>12)||(a!=7)&&(a>15))
```

Składnia „if”

```
if (a>2)  // warunek logiczny 1
{
   // instrukcje 1, np.:
   cout << "a is greater than 2" << endl;
}
else if (a<2)   // warunek logiczny 2
{
   // instrukcje 2, np.:
   cout << "a is smaller than 2" << endl;
}
else   // pozostałe przypadki
{
   // instrukcje w pozostałych przypadkach. np.:
   cout << "a is equal 2" << endl;
}
```

Sprawdzanie poprawności danych strumienia wejściowego

```
cin.fail();
```

Sprawdza poprawność strumienia wejściowego, zwraca PRAWDA, jeżeli operacja wejścia zakończyła się błędem.

```
cin.clear();
```

Resetuje (czyści) stan błędu.

```
cin.ignore();
```

Usuwa (ignoruje) pozostałą zawartość strumienia; jeden znak lub 1000 znaków do końca linii.

```
cin.ignore(1000,'\n');
```

LABORATORIUM

1 (01.10.2025)

2, 3 (08.10.2025)

Zadanie 1

(Wczytywanie i wyświetlanie liczb i tekstów, podstawowe działania)

Napisz prosty program wczytujący liczby (całkowite lub rzeczywiste) z klawiatury, wykonaj proste obliczenia na wczytanych danych (np. dodawanie) i wypisz wynik działania na ekranie z odpowiednim opisem (komentarzem). W programie wykorzystaj deklarację zmiennych liczbowych i tekstowych.

Sprawdź co się stanie jeżeli zamiast liczby wprowadzisz tekst i odwrotnie. Przetestuj podstawowe działania na liczbach tych samych typów (int oraz float) oraz na różnych. Sprawdź co się stanie przy próbie dzielenia przez 0.

Zadanie 2

(Sprawdzanie wieku kandydata na prezydenta)

Napisz program, który zapyta użytkownika o rok urodzenia i na tej podstawie określi jego wiek. W zależności od wieku kandydata program ma sprawdzić czy jest on osoba pełnoletnią (18+) i czy może kandydować na stanowisko prezydenta (35+). W zależności od wieku kandydata program ma wyświetlić odpowiedni komentarz. Można rozbudować program o inne opcje związane z wiekiem.

Zadanie 3

(Rozwiązywanie równania kwadratowego)

Napisz program do rozwiązywania równania kwadratowego o postaci: $\mathbf{ax^2+bx+c=0}$, na podstawie wczytywanych parametrów **a**, **b** i **c**. Zastosuj instrukcję warunkową „if” w celu wykonania odpowiednich obliczeń i komentarzy w zależności od wartości parametru *delta*. W razie potrzeby wykorzystaj bibliotekę `<cmath>`.

WYKŁAD

2

(15.10.2025)

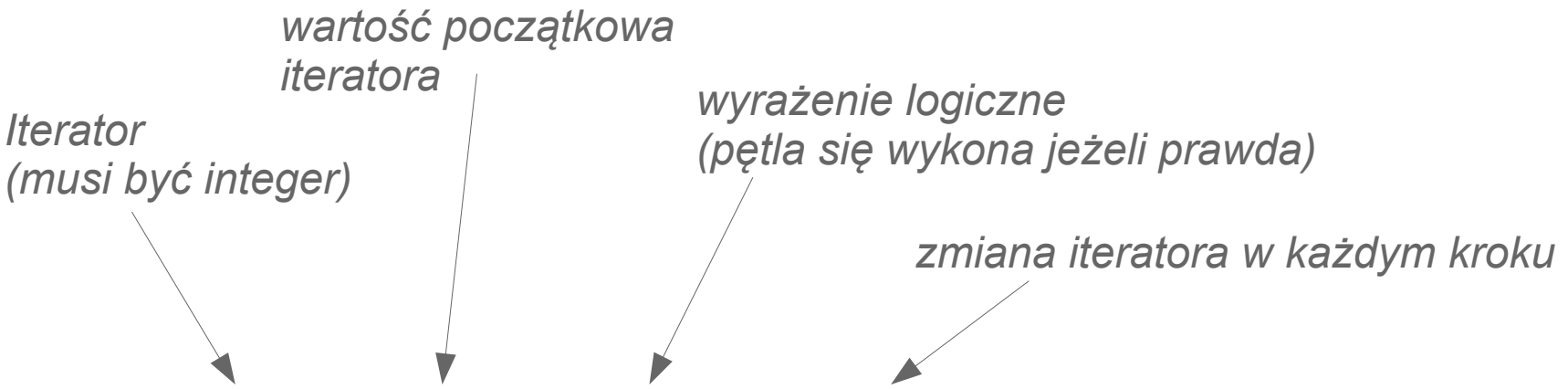
PĘTLE

- **pętla** – instrukcja powtarzalna (iteracyjna)
- **iteracja** – pojedyncze wykonanie pętli
- **iterator** – licznik (przechowuje numer iteracji)

Pętle w C++:

- `for`
- `while`
- `do ... while`

Pętla ”for” (kontrolowana iteratorem)



Inkrementacja: `i++` to samo co `i=i+1` oraz `i+=1`

Dekrementacja: `i--` to samo co `i=i-1` oraz `i-=1`

Pętle ”while”, ”do … while”

(kontrolowane warunkiem logicznym, nie posiadają iteratora)

```
while (log. condition)
{
    instructions;
}
```

Zestaw instrukcji może się nie wykonać ani razu.

```
do
{
    instructions;
} while (log. condition);
```

Zestaw instrukcji wykona się przynajmniej raz.

W obu przypadkach zestaw wykonywanych instrukcji powinien wpływać na zmianę warunku logicznego, który zakończy wykonywanie pętli. W innym przypadku pętle nie skończą się, co może być pożądanym efektem (wyjątkowo).

Zagnieżdżanie pętli (pętla w pętli)

Można zagnieżdżać różne rodzaje pętli wewnątrz innych pętli.

Przykład – podwójna pętla "for":

```
for (int i=1;i<=9;i++)
{
    for (int j=1;j<=9;j++)
    {
        cout<<i<<"-"<<j<<"   ";
    }
    cout<<endl;
}
```

Przerywanie działania pętli

break; //przerwanie wykonywania pętli, jeśli wystąpi w pętli wewnętrznej to tylko ta zostanie przerwana

```
j=1;
while (j<5)
{
     i=1;
     while (i<10)
     {
          cout<<"*";
          if (i==5)
               break;
          i++;
     }
     cout<<endl;
     j++;
}
```

```
i=0;
while (i<5)
{
     i++;
     if (i==3)
          continue;
     cout<<i;
}
```

continue; //przerywa wykonywanie danej iteracji pętli, program przechodzi do kolejnej iteracji (tej samej pętli)

LICZBY (PSEUDO)LOSOWE

- Kompilator GCC posiada wbudowane funkcje generujące liczby losowe (np. rand), ale jak one działają?

- Czas w komputerze (*Unix time*, *POSIX time*) - liczba sekund od 01.01.1970 i cały czas się zmienia

- Jak zamienić tą liczbę (np. 1 234 567 890) na „losową” liczbę, powiedzmy od 1 do 100?

- Zastosować operację reszty z jej dzielenia przez 100; wynik będzie od 0 do 99 (np.:1229 % 100 = 29), potem można dodać 1.

Obliczanie kolejnych liczb losowych

$x_1=(s+b)\%n$ – zaczynamy od liczby zależnej od czasu

$x_2=(s+x_1+b)\%n$

$x_3=(s+x_2+b)\%n$ ... i tak dalej ...

x_i – kolejna liczba (pseudo)losowa

s – Unix time

n – liczba wylosowanych liczb lub zakres (0 ... n-1)

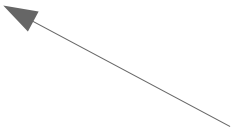
b – jakaś stała (np. 1)

Funkcje losujące liczby naturalne

```
#include<cstdlib>

rand()%n;           // liczba losowa od 0 do n-1
rand()%n+1;         // liczba losowa od 1 do n

// przykladowy zakres od 51 do 75 (25 liczb):
rand()%25;          // liczba losowa od 0 do 24
rand()%25+51;       // liczba losowa od 51 do 75
```



Ile jest liczb całkowitych od 51 do 75

Randomizacja

```
#include<ctime>


srand(time(nullptr)); // rozpoczyna randomizacje z
wykorzystaniem czasu komputera, stosujemy raz w
programie, przed pierwszym uzyciem funkcji rand()


RAND_MAX              // maksymalna liczba losowa
```

INSTRUKCJA WIELOKROTNEGO WYBORU „switch … case”

```
switch (choice)
{
    case 1:
    {
        // instrukcje gdy choice==1
    }
    break;  // koniecznie!
    case 2:
    {
        // instrukcje gdy choice==2
    }
    break;  // koniecznie!
    case 3:
    {
        // instrukcje gdy choice==3
    }
    break;  // koniecznie!
    default:
    {
        // instrukcje gdy choice jest inne niż wyżej
    }
}
```

Składnia

zmienna kontrolna:

int
char
~~**float**~~
~~**string**~~

Użyteczne

```
break;                          // zatrzymuje pętle i przechodzi dalej
exit(0);                // (#include <cstdlib>) kończy program i zwraca 0
system("cls");           //(#include <cstdlib>) czyści terminal (Windows)
system("clear");         //(#include <unistd.h>) czyści terminal (Linux)
getch();   // (#include <conio.h>) czeka na wcisnięcie dowolnego klawisza
```

Nieskończone pętle

```
while(true);
for(;;);
```

Pomiar czasu

```
#include<ctime>

float sum=0, add=1, elapsed;
clock_t start, stop;                          // zmienne clock_t
int iterations=1000*1000*1000;

int main ()
{
    start=clock();                          // start pomiaru czasu
    for (int i=0;i<iterations;i++)
    {
        sum+=add;
        add/=2.0;
    }
    stop=clock();                           // stop pomiaru czasu
    elapsed=float(stop-start)/CLOCKS_PER_SEC;
    cout<<"Time measured: "<<elapsed<<endl;
    return 0;
}
```

LABORATORIUM

4 (15.10.2025)

5, 6 (22.10.2025)

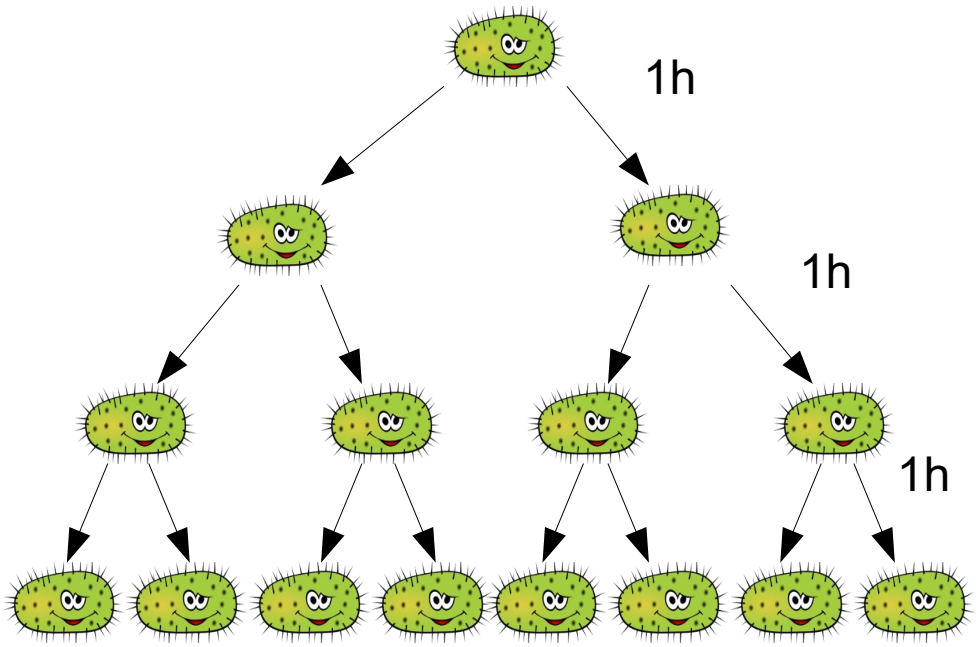
Zadanie 4

Stosując podane poniżej rozwiązania, napisz program, który będzie udawał odliczanie do startu rakiety (z dźwiękiem). Program powinien odliczać od 10 do 1 co 1 sekundę i napisać "START", gdy licznik dojdzie do 0.

```
#include <windows.h>     // windows
#include <unistd.h>      // linux
Sleep(1000)              // czekaj 1000 ms
Beep(2000,500)           // dźwięk (f[Hz], t[ms])
system("cls")            // windows
system("clear")          // linux
```

Zadanie 5

Rysunek przedstawia przyrost populacji bakterii w czasie. Napisz program, który będzie liczył liczebność populacji po upływie kolejnych godzin aż do osiągnięcia 1 000 000 000 organizmów. Zastosuj pętlę "while".




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

Zadanie 6

a) Napisz program, który zapyta użytkownika o liczbę dodatnią. Jeżeli użytkownik poda liczbę ujemną lub zero, program poprosi ponownie o podanie dodatniej liczby. Program powinien powtarzać próbę wczytania dodatniej liczby do momentu, gdy użytkownik taką liczbę poda lub zakończyć działanie po ustalonej liczbie nieudanych prób. Zastosuj pętlę „`do...while`”.

b) Napisz program, który zapyta użytkownika o liczbę rzeczywistą. Jeżeli użytkownik wprowadzi błędną zmienną do strumienia wejściowego to program zapyta ponownie o podanie liczby rzeczywistej. Program ma działać w pętli aż do momentu, gdy użytkownik wprowadzi prawidłową liczbę rzeczywistą lub zakończy działanie po ustalonej liczbie nieudanych prób.

Zadanie 7

Napisz program, który podobnie jak w losowaniu LOTTO wybierze losowo 6 liczb spośród 49 i wypisze je na ekranie. Zastanów się czy program faktycznie działa jak losowanie LOTTO. Jeżeli nie to dlaczego? Popraw wówczas program tak, aby działał jak losowanie LOTTO.

Zadanie 8

Napisz program (grę), który wylosuje liczbę od 1 do 100 i poprosi użytkownika o jej zgadnięcie. Po próbie odgadnięcia program powinien wyświetlić odpowiedni komentarz, np.: „Zgadłeś”, „Za duża” lub „Za mała”. Program powinien pamiętać numer próby i po odgadnięciu wypisać go na ekranie wraz ze słowem „Zgadłeś”. Zastosuj pętlę „while” („do … while”) oraz instrukcję warunkową „if”.

Zadanie 9

Napisać program do generowania liczb rzeczywistych z zakresu, który użytkownik poda z klawiatury. Wykorzystaj metodę generowania liczb całkowitych poznaną na zajęciach w połączeniu z odpowiednimi operacjami matematycznymi.

Zadanie 10

Napisz program – kalkulator, z wykorzystaniem instrukcji wielokrotnego wyboru „switch case”. Program będzie wyświetlał menu główne z dostępnymi operacjami matematycznymi. Spraw, żeby program działał w pętli, dopóki użytkownik nie wybierze opcji WYJŚCIE z menu.

Zadanie 11

Napisz program do obliczania przybliżonej wartości liczby PI metodą Monte Carlo. Algorytm: 1) Wylosuj *n* współrzędnych punktów (par liczb) wewnątrz kwadratu o boku 2*r* (*r*-dowolne) i środku w początku układu współrzędnych. 2) Sprawdź ile z nich (*m*) mieści się wewnątrz okręgu o promieniu *r* i środku w początku układu współrzędnych. 3) Zauważ, że stosunek pól tych figur powinien być taki jak stosunek odpowiednich ilości punktów wewnątrz kwadratu i okręgu i można z niego łatwo policzyć wartość PI. Spraw, aby program działał w pętli licząc PI dla liczb *n*=10, 100, 1000 ..., czyli dla kolejnych potęg 10. Ustaw liczbę tych pętli (potęg) tak, aby czas działania programu nie przekraczał minuty (około). Na ekranie wypisuj: nr iteracji, liczbę wylosowanych punktów *n,* obliczone PI, względną różnicę między rzeczywistą liczbą PI a obliczoną, czas trwania iteracji.

```
        OBLICZANIE PI METODA MONTE CARLO
================================================
1   10   3.6   14.5916 %   0 sec.
2   100   3.24   3.1324 %   0 sec.
3   1000   3.2   1.85917 %   0 sec.
4   10000   3.1336   0.254414 %   0 sec.
5   100000   3.1388   0.0888959 %   0.004 sec.
6   1000000   3.1421   0.0162738 %   0.012 sec.
7   10000000   3.14148   0.00362481 %   0.173 sec.
8   100000000   3.14149   0.00312393 %   1.668 sec.
9   1000000000   3.14152   0.00225118 %   16.81 sec.
```

WYKŁAD

3

(29.10.2025)

TABLICE (MACIERZE)

Tablica – zmienna złożona reprezentująca uporządkowany (ponumerowany) zestaw zmiennych jednego typu

deklaracja: `int dane[10];`

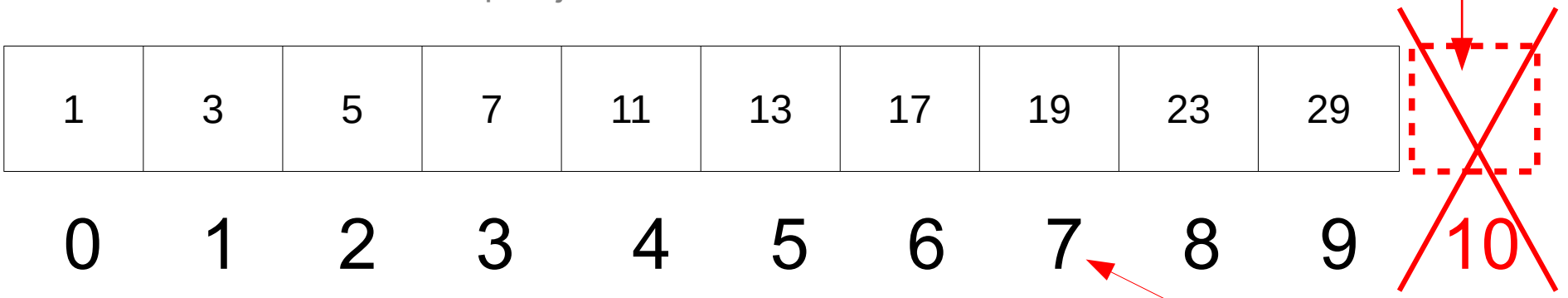
nawiasy kwadratowe są zarezerwowane w C++ wyłącznie dla tablic

typ danych

nazwa tablicy

rozmiar tablicy,w przypadku tablicy statycznej jej rozmiar musi być wartością stałą, znaną na etapie kompilacji

ostrożnie z wywoływaniem tej zmiennej !!!



numeracja: od **0** do **rozmiar-1**

index

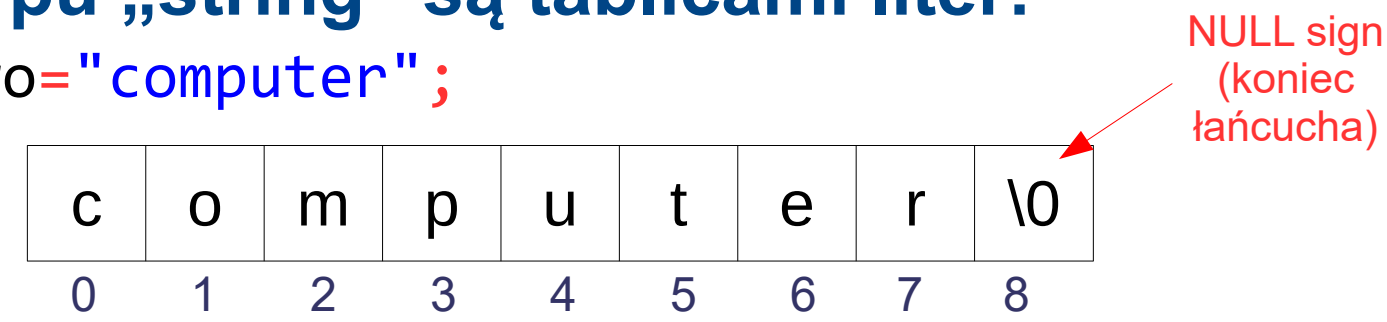
index – unikalny numer miejsca w tablicy (pozycja)

Przykład użycia (odwołania do elementu):

```
cout<<dane[3];  //wypisz element o indeksie 3
cin>>dane[i];   //zapisz do elementu o indeksie i
```

Zmienne typu „string” są tablicami liter:

```
string slowo="computer";
```




```
cout<<slowo[3];      //wypisze literę "p" na ekranie
```

Wielowymiarowe tablice:

```
float macierz[5][4];                    //macierz 2D
double dane[10][10][10][10][10];        //macierz 5D
```

ZAPISYWANIE DO PLIKU TEKSTOWEGO

```
#include<fstream>          //biblioteka odpowiedzialna za pracę z plikami

fstream plik1;             //tworzenie obiektu plik1 klasy fstream

                           //tryb zapisu | dodawanie zawartości
plik1.open("filename.txt",ios::out|ios::app);
        //otwiera plik w aktualnej lokalizacji, jeżeli nie istnieje – będzie utworzony

plik1<<x<<"   "<<y<<endl;
                  //zapisanie zmiennych x i y rozdzielonych 3 spacjami do pliku
                          (kierowanie strumienia wyjściowego do obiektu plik1)
plik1.close();
              //zamykanie pliku, ważne, zawsze zamknij plik jeżeli został otwarty !
```

OBIEKTOWOŚĆ !!!

open, ***close***, ***good***, ***eof*** są metodami pracującymi na obiekcie ***plik1*** klasy ***fstream***

CZYTANIE Z PLIKU TEKSTOWEGO

```
plik1.open("filename.txt",ios::in);          //tryb odczytu

plik1.good();       // (prawda, jeżeli otwieranie pliku powiodło się,
                    // fałsz, jeśli nie lub plik nie istnieje)

plik1.eof(); // prawda, jeżeli podczas czytania osiągnięto koniec pliku

plik1>>a;           // czytanie jednej zmiennej z pliku, zapisywanie
                    // jej na zmienną a, (kierowanie strumienia
                    // wyjściowego z obiektu plik1 na zmienną a)

plik1>>a>>b>>c;              //czytanie kolejno 3 zmiennych z pliku,
                             // zapisywanie ich na zmienne a, b, c,
                             // i przejście do następnej linii

                  // Powyższą metodę najczęściej stosujemy gdy wczytujemy
                  // dane uporządkowane o znanym typie.

plik1.close();               //pamiętaj, żeby zamknąć plik po odczycie
```

```
#include<cstdlib>

getline(plik1,linia); //czyta całą linię z plik1 (bez względu na
                      // to co się w niej znajduje), zapisuje na
                      // zmienną linia typu string i zwraca prawdę

atoi(linia.c_str());  //przekształca string na int lub float

atof(linia.c_str());
```

Metoda c_str() konwertuje ciąg znaków zapisany w zmiennej typu string na ciąg który może być zapisany w tablicy znaków.

```
ifstream fin("plik.txt");        // tylko czytanie z pliku
ofstream fout("plik.txt");       // tylko pisanie do pliku
fstream   file("plik.txt");      // czytanie i/lub pisanie

// używając fstream należy jawnie podać tryb otwarcia, np.:
ios::in                                  // czytanie
ios::out                                 // pisanie
ios::app                                 // dopisywanie
ios::binary                              // tryb binarny

fstream f1("plik.txt",ios::in);          // tylko czytanie
fstream f2("plik.txt",ios::out);         // tylko pisanie
fstream f3("plik.txt",ios::in|ios::out);
                                         // czytanie i pisanie
fstream f4("plik.txt",ios::out|ios::app);
                                         // tryb dopisywania
```

Manipulacja strumieniem wyjściowym (formatowanie zapisu)

```
#include<iomanip>

cout<<setprecision(6);                    // 6 liczb znaczących
cout<<fixed<<setprecision(6);             // 6 liczb po przecinku

cout.width(10);                           // szerokość 10 znaków
cout<<setw(10)<<12.3;

cout<<left<<12.3<<endl;                   // wyrównanie do lewej
cout<<right<<12.3<<endl;                  // wyrównanie do prawej

cout.unsetf(std::ios_base::floatfield);   // wyłączenie fixed
```

Otwieranie pliku o nazwie podanej przez użytkownika

```
#include <fstream>                    // operacje na plikach
#include <string>

string filename;
ifstream file;                        // plik do odczytu
cout << "Enter the file name: ";
cin >> filename;
file.open(filename);                  // otwórz plik

if (!file) {
  cout << "Error: could not open file '";
  cout << filename << "'" << endl;
  return 1;                           // zakończ z kodem błędu
}

cout << "File '" << filename << "' opened" << endl;
// czytanie z pliku
file.close();
```

ZŁOŻONE TYPY DANYCH
(1) KLASA STRING

Z **obiektów klasy** `string` już korzystamy, są wygodniejsze niż łańcuchy znaków (`c_str` lub `char`). Po dołączeniu biblioteki `<string>` można wykonywać takie opreracje jak: konkatenacja (+), porównywanie (==,<), jak również korzystać z **metod** takich jak:

```
• size(),length()          // zwracają długość łańcucha
• empty()                  // sprawdza czy jest pusty
• append("abc")            // dodaje tekst na końcu
• substr(pos,len)          // zwraca fragment łańcucha
• find("c")                // zwraca pierwszą pozycję frazy
• replace(pos,len,str)     // zamiana fragmentu łańcucha innym
• erase(pos,len)           // usuwa fragment lańcucha
• insert(pos,str)          // wstawia tekst w podanym miejscu
• at(i)                    // zwraca znak na pozycji i
• c_str()       // zwraca wskaźnik do klasycznego c_string -a
```

```
string c;          // przykład
c.substr(2,3);
```

obiekt.metoda

```
#include<string>  // C++ (operacje na obiektach klasy string)
#include<string.h>           // C (operacje na tablicach char

string napis1;          // obiekt napis1 klasy string
char napis2[20];        // tablica znaków typu char

napis1="Ala ma kota 1";
strcpy(napis2,”Ala ma kota 2”);

cout<<napis1<<endl;
cout<<napis2<<endl;
```

Przechowywanie w pamięci

Przechowywanie w RAM – tablica znaków (char):

znak:	A	l	a		m	a		k	o	t	a	\0
index:	0	1	2	3	4	5	6	7	8	9	10	11
ASCII:	65	108	97	32	109	97	32	107	111	116	97	0

Pierwszy znak ma zawsze index=0

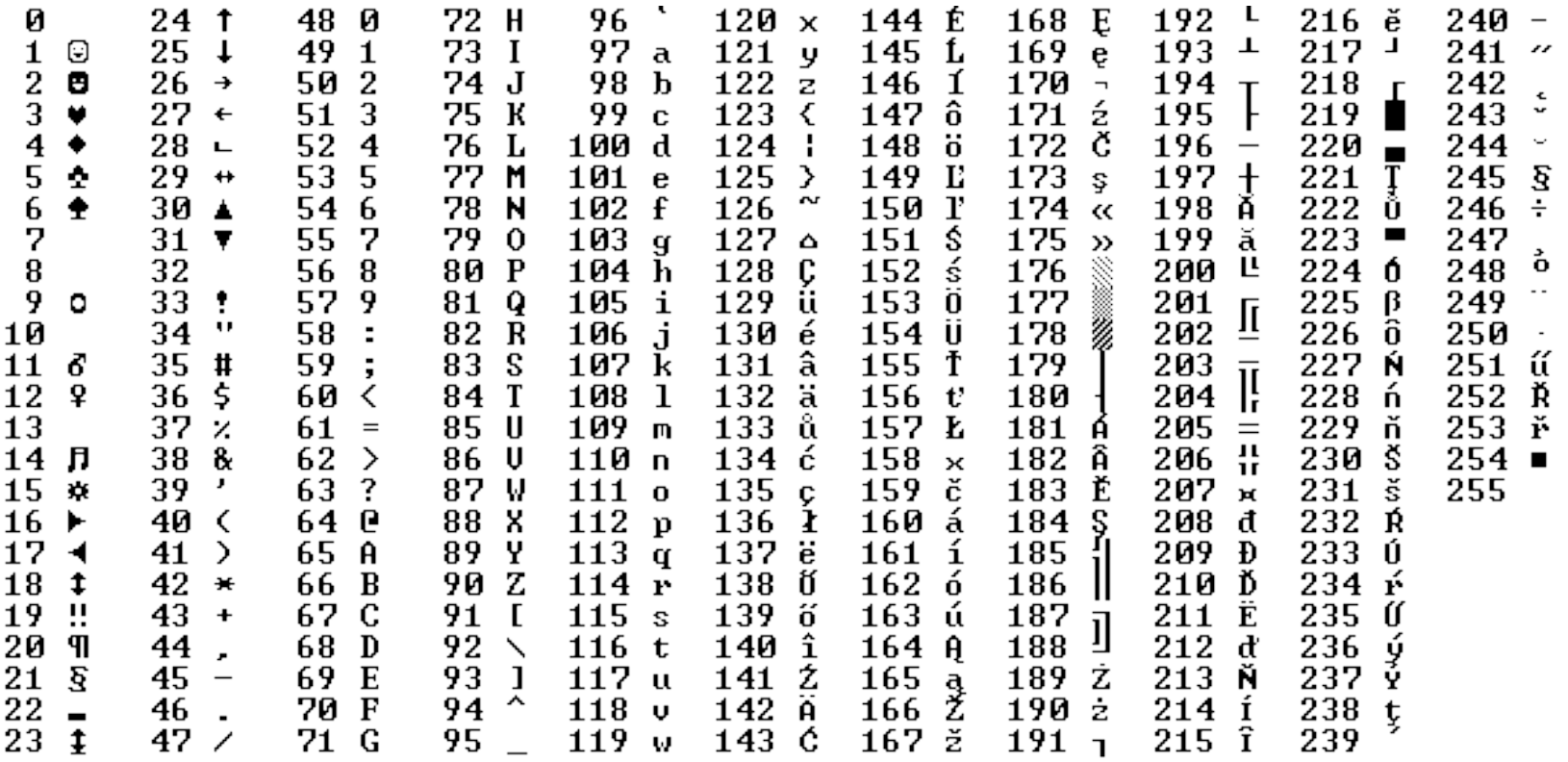
NULL sign kończy łańcuch

```
cout<<napis1[4]<<endl;
cout<<napis2[4]<<endl;
```

Do elementów, które są znakami typu char dostajemy się w ten sam sposób.

ASCII table

American Standard Code for Information Interchange



0		24	↑	48	0	72	H	96	`	120	x	144	É	168	Ę	192	└	216	ě	240	-
1	☺	25	↓	49	1	73	I	97	a	121	y	145	Ĺ	169	ę	193	┴	217	┘	241	˝
2	☻	26	→	50	2	74	J	98	b	122	z	146	Í	170	¬	194	┬	218	┌	242	˛
3	♥	27	←	51	3	75	K	99	c	123	{	147	ô	171	ź	195	├	219	█	243	ˇ
4	♦	28	∟	52	4	76	L	100	d	124	¦	148	ö	172	Č	196	─	220	▄	244	˘
5	♣	29	↔	53	5	77	M	101	e	125	}	149	Ľ	173	ş	197	┼	221	Ţ	245	§
6	♠	30	▲	54	6	78	N	102	f	126	~	150	ľ	174	«	198	Ă	222	ů	246	÷
7		31	▼	55	7	79	O	103	g	127	⌂	151	Ś	175	»	199	ă	223	▀	247	¸
8		32		56	8	80	P	104	h	128	Ç	152	ś	176	░	200	Ľ	224	Ó	248	°
9	○	33	!	57	9	81	Q	105	i	129	ü	153	Ö	177	▒	201	╔	225	ß	249	¨
10		34	"	58	:	82	R	106	j	130	é	154	Ü	178	▓	202	╩	226	Ô	250	·
11	♂	35	#	59	;	83	S	107	k	131	â	155	Ť	179	│	203	╦	227	Ń	251	ű
12	♀	36	$	60	<	84	T	108	l	132	ä	156	ť	180	┤	204	╠	228	ń	252	Ř
13		37	%	61	=	85	U	109	m	133	ů	157	Ł	181	Á	205	═	229	ň	253	ř
14	♫	38	&	62	>	86	V	110	n	134	ć	158	×	182	Â	206	╬	230	Š	254	■
15	☼	39	'	63	?	87	W	111	o	135	ç	159	č	183	Ě	207	¤	231	š	255	
16	►	40	(	64	@	88	X	112	p	136	ł	160	á	184	Ş	208	đ	232	Ŕ		
17	◄	41	)	65	A	89	Y	113	q	137	ë	161	í	185	╣	209	Đ	233	Ú		
18	↕	42	*	66	B	90	Z	114	r	138	Ő	162	ó	186	║	210	Ď	234	ŕ		
19	‼	43	+	67	C	91	[	115	s	139	ő	163	ú	187	╗	211	Ë	235	Ű		
20	¶	44	,	68	D	92	\	116	t	140	î	164	Ą	188	╝	212	ď	236	ý		
21	§	45	-	69	E	93	]	117	u	141	Ź	165	ą	189	Ż	213	Ň	237	Ý		
22	▬	46	.	70	F	94	^	118	v	142	Ä	166	Ž	190	ż	214	í	238	ţ		
23	↨	47	/	71	G	95	_	119	w	143	Ć	167	ž	191	┐	215	Î	239	´		

duże litery (65-90) + 32 = małe litery (97-122)

```
cout<<char(4)<<endl;  // wypisywanie znaków z tablicy ASCII
```

Wypiszmy numery elementów, znaki, numery kodów ASCII

```
for(int i=0;i<=napis1.length();i++)
    cout<<i<<" "<<napis1[i]<<" "<<int(napis1[i])<<endl;
cout<<endl;
for(int i=0;i<=20;i++)
    cout<<i<<" "<<napis2[i]<<" "<<int(napis2[i])<<endl;
```

```
int l=napis1.length();

// metoda length działająca na obiekt klasy string
(obiektowość) zwraca długość zmiennej lub numer
ostatniego znaku (null sign)
```

Wczytywanie tekstu/znaku z klawiatury

"a"= | a | \0 |

```
//cudzysłów gdy łańcuch znaków
```

'a'= | a |

```
//apostrof gdy pojedyncza litera
```

W strumieniu wejściowym spacja jest traktowana jako separator !

```
cin>>napis;
getline(cin,napis);
```

```
//użycie getline zamiast cin pozwala czytać z klawiatury napisy zawierające spacje aż do znaku następnej linii (enter, ¶)
```

Konkatenacja – łączenie

```
string jeden="Ala ma ";
string dwa="kota";
string trzy=jeden+dwa;
```

Znajdowanie pozycji frazy

```
string napis="Ala ma kota";
string szuk="ma";
int position=napis.find(szuk);
                        //jeżeli brak to position=-1
```

```
napis.erase(3,5);
                          //usuwa 5 znaków zaczynając od 3

napis.insert(11,"dostaw");
                    //wstawia string "dostaw" od pozycji 11

napis.replace(4,2,"zastepstwo");
   //zastępuje 2 znaki tekstem "zastepstwo" od pozycji 4

string nowynapis=napis.substr(4,7);
    //wycina nowy tekst z 7 kolejnych znaków, ze starego
                                        zaczynając od pozycji 4
```

Zamiana znaków małe ↔ DUŻE

```
#include<algorithm>

transform(napis.begin(),napis.end(),napis.begin(),::toupper);

transform(napis.begin(),napis.end(),napis.begin(),::tolower);
```

Procedura zmiany zmiennej typu string na tablicę znaków typu char

```
#include<cstring>

string napis="Ala ma kota";
int n=napis.length();   // zwraca długość łańcucha
char tab_char[n];          // tworzy tablicę znaków
strcpy(tab_char,napis.c_str());     // przepisanie


//funkcja strcpy konwertuje ciąg znaków zapisany w obiekcie
klasy string (metoda c_str() zwraca wskaźnik do tablicy
znaków zakończonej znakiem NULL) na ciąg który może być
zapisany w tablicy znaków (tab_char), może być potrzebne gdy
chcemy korzystać z funkcji języka C jub jego starszych
procedur i funkcji
```

LABORATORIUM

7 (29.10.2025)

8, 9 (05.11.2025)

Zadanie 12 (Prosta analiza danych)

wersja podstawowa:

Napisz program obliczający średnią i odchylenie standardowe liczb wczytywanych z klawiatury (rzeczywiste). Wykorzystaj zmienną tablicową do przechowania danych wejściowych. Obliczenia wykonaj dopiero po wczytaniu tej właśnie tablicy danych.

wersja rozszerzona:

Napisz program, który: (1) zapyta użytkownika o ilość (*n*) liczb rzeczywistych do wylosowania oraz zakres, z którego mają być losowane, (2) wylosuje te liczby i zapisze je w tablicy, (3) znajdzie lub obliczy i wyświetli następujące wartości: najmniejszą, największą, sumę, średnią, odchylenie standardowe, medianę, dominantę.

Zadanie 13 (Sortowanie liczb)

Napisz program, który sortuje dużą tablicę liczb w porządku rosnącym lub malejącym, wykorzystując dwie proponowane metody (algorytmy): sortowanie bąbelkowe (bubble sort) i sortowanie szybkie (quick sort).

Program powinien: (1) Poprosić użytkownika o podanie rozmiaru tablicy. (2) Wylosować tablicę liczb rzeczywistych (lub całkowitych) o podanym rozmiarze. (3) Znaleźć i wyświetlić największy oraz najmniejszy element tablicy. (4) Posortować tablicę dwoma metodami (bąbelkową i szybką), mierząc czas wykonania każdej z nich. Po zakończeniu sortowania wypisać na ekranie: czas sortowania dla każdej metody, pięć pierwszych oraz pięć ostatnich elementów: tablicy wylosowanej, tablicy posortowanej metodą bąbelkową, tablicy posortowanej metodą szybką. Wyniki należy przedstawić np. w formie tabeli z trzema kolumnami:

```
Wylosowane     Bubble sort    Quick sort
=========================================
```

Zadanie 14 (Prosta baza danych)

Napisz program, który obsługuje prostą bazę danych wczytywaną z pliku tekstowego. Plik ma strukturę powtarzających się trójek linii: 1. numer rekordu (liczba całkowita), 2. nazwa (np. imię lub imię i nazwisko), 3. jakaś liczba (np. rok urodzenia).
Program powinien wczytać wszystkie rekordy do tablic (np. trzy równoległe tablice: id[], name[], year[]), a następnie umożliwić użytkownikowi wykonywanie operacji z menu opartego na instrukcji switch-case. Menu programu ma zawierać następujące opcje:

0. Wyjdź z programu — zakończ działanie.
1. Wyświetl wszystkie rekordy – wypisz dane w czytelnym, uporządkowanym formacie.
2. Dodaj nowy rekord – dopisz rekord na koniec tablic oraz do pliku.
3. Usuń rekord (dla chętnych) – usuń rekord o podanym numerze i zaktualizuj plik.

Program powinien działać w pętli, aż użytkownik wybierze opcję zakończenia. Wczytywanie i zapisywanie danych należy zrealizować z użyciem bibliotek <fstream> oraz tablic do przechowywania rekordów.

LABORATORIUM

10 (12.11.2025)

11, 12 (19.11.2025)

Zadanie 15 (Czytanie i wyświeltlanie danych)

Struktura pliku tekstowego z danymi jest następująca: pierwsze trzy linie to komentarze – dwie zawierają informacje tekstowe o danych, a trzecia określa liczbę rekordów.

W kolejnych liniach znajdują się dane liczbowe (rzeczywiste) ułożone w czterech kolumnach: wielkość fizyczna X, jej niepewność dX, wielkość fizyczna Y, jej niepewność dY (oddzielone spacjami lub tabulatorami).

Przygotuj taki plik samodzielnie, korzystając z Notatnika lub arkusza kalkulacyjnego. Pamiętaj, aby liczby zmiennoprzecinkowe zapisywać z kropką, zgodnie z notacją angielską.

Napisz program, który: zapyta użytkownka o nazwę pliku z danymi, wczyta cały plik (dane liczbowe do jednej tablicy), wypisze na ekran pierwsze trzy linie komentarza, po czym zapyta użytkownika, czy chce wyświetlić wszystkie dane. Jeśli użytkownik potwierdzi – program wypisze pełną zawartość tablicy danych, jeżeli nie – zakończy działanie.

Zadanie 16 (Prosta analiza tekstu)

Napisz program, który poprosi użytkownika o podanie sentencji (tekstu) do analizy. Program policzy z ilu znaków (łącznie ze znakami niedrukowalnymi) i wyrazów składa się tekst. Następnie program poprosi użytkownika o podanie litery do wyszukania w ww sentencji. Program policzy ile określonych w zapytaniu liter (np. 'a' lub ‘A’) jest w podanym z klawiatury tekście.

Wielkość liter nie ma znaczenia, zliczamy zarówno duże jak i małe litery. Wynik wypisujemy na ekranie. Proszę zwrócić uwagę na możliwość podania dużej lub małej litery, podczas gdy program ma znaleźć ich liczbę niezależnie od tego, czy w zdaniu występuje litera mała czy duża.

Zadanie 17 (Szyfr Cezara)

Opracuj prosty schemat szyfrowania oparty na numerach znaków z tablicy ASCII, np. przesunięcie numeru znaku o jeden lub więcej ("szyfr Cezara"). Osoba szyfrująca* tworzy plik *sentence.txt* zawierający zdanie do zaszyfrowania oraz pisze program, który odczyta to zdanie, zaszyfruje za pomocą klucza (iczba przesunięcia) a następnie zapisze zaszyfrowane zdanie do pliku *encrypted.txt*. Osoba odszyfrowująca* znając klucz szyfru pisze program, który odczyta zaszyfrowane zdanie, odszyfruje a odszyfrowane zdanie zapisze do pliku *decrypted.txt*. Sukcesem jest identyczna zawartość (co do treści) plików *sentence.txt* i *decrypted.txt*.

* Osoba szyfrująca i odszyfrowująca może być jedną i tą samą osobą w celu wykonania zadania.

WYKŁAD

4

(26.11.2025)

FUNKCJE

- **Funkcja** – podprogram, wydzielony fragment kodu, wykonujący określone zadanie, który może być wielokrotnie wywoływany. Zwykle przyjmuje argumenty (ale nie musi), wykonuje operacje i może zwracać wartość (ale nie musi). Dzięki funkcjom program staje się czytelniejszy, uporządkowany i łatwiejszy w utrzymaniu. Jest podstawowym elementem paradygmatu programowania proceduralnego
- **Programowanie proceduralne** – paradygmat programowania polegający na podziale kodu na fragmenty (procedury, funkcje) wykonujące określone zadania. Jest rozszerzeniem paradygmatu programowania strukturalnego kładącym dodatkowy nacisk na modularność i wielokrotne użycie kodu poprzez zastosowanie funkcji.
- **Programowanie strukturalne** - paradygmat programowania polegający na pisaniu programów w sposób uporządkowany, z wykorzystaniem trzech podstawowych struktur: sekwencji (kolejno), warunków (logicznych) i pętli. Program dzieli się na mniejsze, czytelne fragmenty (np. funkcje) a przepływ sterowania odbywa się w sposób przewidywalny – bez skoków typu goto. Dzięki temu kod jest prostszy, bardziej czytelny i łatwiejszy do utrzymania.

- Funkcja główna w C++ (`int main()`) – zarządza pozostałymi funkcjami (jeżeli istnieją)
- Deklaracja funkcji może, ale nie musi zawierać nazw argumentów formalnych, może być po prosu kopią definicji (nagłówka).
- Do funkcji wysyłane są kopie wartości argumentów poprzez argumenty formalne, funkcja nie ma dostępu do oryginalnych zmiennych (aktualnych argumentów), tylko do ich wartości. Wszelkie operacje są wykonywane na kopii i nie są widoczne poza blokiem funkcji.
- Instrukcja return oznacza powrót z funkcji do miejsca jej wywołania. Wykonanie instrukcji `return` powoduje zakończenie wykonywania funkcji (również `main()`). Jeśli funkcja powinna coś zwrócić to zwracana rzecz (wartość lub zmienna) pojawia się po słowie `return`. W obrębie jednej funkcji może pojawić się więcej niż jedna taka instrukcja. Jeżeli funkcja nic nie zwraca to po `return;` jest średnik lub pomijamy `return`.

Przykład 1
(Funkcja do obliczania potęgi)

```
#include<iostream>

using namespace std;

float num=2.3;          //zmienne globalne (każda funkcja ma dostęp)
int pow=4;

float power(float,int);    //deklaracja funkcji; typ wyniku, nazwa,
                        typy parametrów wejściowych, nawiasy okrągłe
int main()                                  //funkcja sterująca (główna)
{           //argumenty aktualne
    cout<<power(num,pow)<<endl;                     //wywołanie funkcji
    return 0;
}
           //argumenty formalne (może być bez)
float power(float a,int n)              //definicja funkcji (nagłówek)
{
    float b=a;             //ciało funkcji; zmienne lokalne, operacje
    for(int i=2;i<=n;i++)
        b*=a;
    return b;                                       //zwracana wartość
}
```

Przykład 2
(Funkcja do wyświetlania menu)

```
// umieszczenie definicji funkcji i jej ciała przed main() - nie trzeba
// wcześniej deklarować, ale jak jest więcej takich funkcji to robi się
// niepraktyczne (kwestia subiektywna)

void main_menu()          //void – nie zwraca nic (kiedyś procedura)
{
    cout<<"   TYTUL PROGRAMU   "<<endl;
    cout<<"====================="<<endl;
    cout<<" Wybierz opcje:"<<endl;
    cout<<"  1. Zrob to"<<endl;
    cout<<"  2. Zrob tamto"<<endl;
    cout<<"  3. Wyjscie"<<endl;
    // nie ma return, może być, ale bez wartości zwracanej
}

int main()
{
    main_menu();                                  // wywołanie funkcji
    // reszta programu
    return 0;
{

// Można się zastanowić, czy warto, żeby funkcja menu() zwracała
// coś, np. liczbę odpowiadającą wybranej opcji.
```

Przykładowe deklaracje

```
double fun1(double);
      //zwraca liczbę double, przyjmuje liczbę double

float fun2(float x,float y);
   //zwraca liczbę float, przyjmuje dwie liczby float

void fun3(float x,int y);
//niczego nie zwraca, przyjmuje liczbę float oraz int

void fun4();
         //niczego nie zwraca,  niczego nie przyjmuje
```

Przekazywanie tablic do funkcji

Tablice są przekazywane do funkcji inaczej niż standardowe zmienne. W przypadku tablic do funkcji zawsze trafia oryginalna zawartość tablicy, co oznacza, że jeśli funkcja zmieni coś w tablicy, to po zakończeniu jej wykonywania ta zmiana będzie zmianą trwałą.

Szczegółowo mechanizm przekazywania tablic do funkcji zostanie przedstawiony podczas omawiania wskaźników. Teraz musi wystarczyć nam wiedza o tym jak stworzyć funkcję przyjmującą jako parametr tablicę i jak taką funkcję wywołać.

Przekazując tablicę do funkcji nie podajemy jej rozmiaru, ale możemy np:

```
void WypiszTab(int tab[5],int rozmiar)
```

Kompilator nie bierze pod uwagę liczby która pojawia się w nawiasach kwadratowych przy nazwie tablicy.

Przeanalizuj poniższe przykłady:

```
void WypiszTab(int tab[],int rozmiar)
{
    for(int i=0;i<rozmiar;i++)
        cout<<tab[i]<<" ";
    cout<<endl;
}

void PowiekszTab(int tab[],int rozmiar)
{
    for(int i=0;i<rozmiar;i++)
        tab[i]++;           //zwiekszamy elementy tablicy o 1
}

int main()
{
    const int r=7;
    int t[r]={1,2,3,4};  //reszta zostanie wypełniona zerami
    WypiszTab(t,r);
    PowiekszTab(t,r);
    WypiszTab(t,r);
    return 0;
}
```

Wartości domyślne parametrów funkcji

Parametrom funkcji można przypisać wartości domyślne. Jeśli parametr ma przypisaną wartość domyślną, to podczas wywoływania tej funkcji nie musimy podawać jego wartości, w takiej sytuacji będzie on miał przypisaną wartość domyślną. Jeśli jednak podamy wartość dla danego parametru, to przyjmie on taką wartość jak została podana podczas wywołania funkcji. Przykład:

```
void NapiszTekst(string napis,int ile=5)
{
    for(int i=0;i<ile;i++)
        cout<<napis<<endl;
}

int main()
{
    NapiszTekst("Witaj",2);
    NapiszTekst("Hello");
    NapiszTekst("Czesc",1);
    return 0;
}
```

Przeciążanie funkcji

W języku C++ możemy tworzyć funkcje o takich samych nazwach, ale muszą się one w takim przypadku różnić typem parametrów lub ich liczbą. Dopuszczalna jest sytuacja, w której funkcje mają taki sam typ parametrów ale w innej kolejności. Przykłady:

```
float dodaj()
float dodaj(int i)
float dodaj(float a, double b)
float dodaj(double a, float b)
```

Podczas wywołania funkcji kompilator wybierze jedną z nich na podstawie liczby i typu argumentu(ów).

W przypadku przeciążania nazw funkcji kompilator nie bierze pod uwagę zwracanego typu. Poniższe dwie funkcje są nieprawidłowe, ponieważ mają taki sam typ parametru:

```
int dodaj(float i)
float dodaj(float i)
```

Sufiksy danych

Sufiksy danych to dodatkowe litery dopisane na końcu literałów liczbowych, które określają typ danych, jaki ma mieć wartość zapisana w kodzie. Dzięki nim możesz jednoznacznie wskazać, czy liczba ma być typu int, long, float, double itd. Jeśli chcemy poinformować kompilator, że dana liczba jest typu float to możemy dodać po liczbie sufix "f" np.:

`dodaj(2.4,3f)` lub `dodaj(2.4f,3.0)`

Domyślnie liczba z kropką jest typu `double`, bez kropki – `int`.

Przykładowe sufiksy dla liczb całkowitych:

```
U,u     -   unsigned int              np. 10u
L,l     -   long int                  np. 20l
LL,ll   -   long long int             np. 23ll
ul      -   unsigned long int         np. 51ul
ull     -   unsigned long long int    np. 47ull
```

Przykładowe sufiksy dla liczb zmienoprzecinkowych:

```
f       -   float                     np. 3.14f
L,l     -   long double               np. 2.71L
```

Przypomnienie
(Zmienne lokalne i globalne)

W funkcjach możemy zadeklarować nowe zmienne, będą one **zmiennymi lokalnymi**. Oznacza to, że będą dostępne tylko w tej funkcji, w której zostały zadeklarowane. Inne funkcje nie mają do nich dostępu. W różnych funkcjach możemy deklarować zmienne o takich samych nazwach. Zmienna lokalna nie ma przypisanej żadnej konkretnej wartości początkowej. Zmienne tworzone w obrębie danego bloku (np. funkcji) są przechowywane w pamięci tylko w momencie wykonywania tego bloku. Po jego zakończeniu, wszystkie zmienne w nim utworzone zostają z pamięci usunięte.

Zmienne globalne to zmienne zadeklarowane poza jakąkolwiek funkcją, również poza funkcją `main()`. Są dostępne dla wszystkich funkcji. Jeśli jakaś funkcja zmieni wartość zmiennej globalnej, to w innej funkcji taka zmiana jest widoczna. Wynika to z tego, że zmienne globalne są przechowywane w pamięci już po uruchomieniu programu i cały czas znajdują się w tym samym miejscu w pamięci (są przechowywane pod tym samym adresem). Wartość domyślna zmiennej globalnej to zero.

Przesłanianie zmiennych

Zmienne globalne mogą być **przesłonięte**, jeśli wewnątrz funkcji (lub bloku) zadeklarujemy inną zmienną o tej samej nazwie (choć niekoniecznie tym samym typie). Wówczas nazwa tej zmiennej w ciele funkcji odnosi się do zmiennej lokalnej. Zmienna globalna istnieje, ale jest w zakresie funkcji (bloku) niewidoczna.

Rekurencja (rekursja)

Rekurencja to technika programowania, w której funkcja wywołuje samą siebie, rozwiązując problem przez podzielenie go na mniejsze, podobne podproblemy. Każde wywołanie działa niezależnie i czeka na wynik kolejnego. Rekurencja zawsze musi mieć warunek brzegowy, który zatrzymuje dalsze wywołania, aby uniknąć nieskończonej pętli.

Przykładem algorytmu, w którym możemy zastosować rekurencję jest obliczanie silni. Po podaniu przez użytkownika liczby całkowitej wywoływana jest funkcja obliczająca silnię i wypisywany jest wynik, który ta funkcja zwróciła.

Innym przykładem algorytmu rekurencyjnego jest algorytm sorotwania szybkiego (quicksort). Dzieli on tablicę na dwie mniejsze części względem elementu pivot (element osiowy, podziału), a następnie rekurencyjnie wywołuje samego siebie do posortowania każdej z tych części, aż do osiągnięcia warunku brzegowego, kiedy podtablica ma 0 lub 1 element.

Przykład 3
(Rekurencyjne obliczanie silni)

```
int silnia(int n)
{
  if(n<=1)
    return 1;
  return n*silnia(n-1);
}

int main()
{
  int a;
  cout<<"Podaj liczbę:"<<endl;
  cin>>a;
  cout<<a<<"!="<<silnia(a)<<endl;
  return 0;
}
```

Przykład 4
(Funkcja do sortowania szybkiego)

```
void quicksort(int tab[], int lewy, int prawy) {
  int i = lewy;
  int j = prawy;
  int pivot = tab[(lewy + prawy) / 2];    // element osiowy (wartość)

  while (i <= j) {
    while (tab[i] < pivot) i++;           // szukaj elementu większego
    while (tab[j] > pivot) j--;           // szukaj elementu mniejszego

    if (i <= j) {
      swap(tab[i], tab[j]);               // zamiana
      i++;
      j--;
    }
  }

  if (lewy<j) quicksort(tab,lewy,j);      // rekurencja dla L. części
  if (i<prawy) quicksort(tab,i,prawy);    // rekurencja dla P. części
}
```

LABORATORIUM

13 (26.11.2025)

14, 15 (03.12.2025)

Zadanie 18 (Obliczanie potęgi)

Rozbuduj funkcję przedstawioną jako przykład do obliczania potęgi w taki sposób, żeby pozwalała na obliczenie również ujemnej potęgi danej liczby. Zadbaj o to, żeby program liczył tylko całkowite potęgi, nawet przy przypadkowym podaniu potęgi jako liczby zmiennoprzecinkowej. Niech program napisze jakie działanie wykonał i jaki jest jego wynik, np.: `(-2.1)^3 = -9.261`

Zadanie 19 (Kalkulator z funkcjami)

Przekształć program KALKULATOR (**Zadanie 10**) tak, aby używał funkcji do wykonywania działań matematycznych i wyświetlania menu głównego. Dodaj działanie potęgowania i obliczania silni. Program powinien działać w pętli aż do wyboru opcji wyjścia.

Zadanie 20 (Histogram)

Napisz program, który wylosuje tablicę n (docelowo duże) liczb z danego zakresu, Program zapyta użytkownika o n oraz granice zakresu. Znajdź i wypisz najmniejszy i największy element tablicy oraz ile wylosowano liczb.

W kolejnym etapie program zapyta użytkownika na ile podprzedziałów chce podzielić przedział losowania. Program policzy i wypisze na ekranie granice podprzedziałów oraz ile liczb znajduje się w każdym z podprzedziałów.

Tam, gdzie to właściwe – pisz i używaj funkcji.

Zadanie 21 (Sortowanie z funkcjami)

Przerób program z **Zadania 13** tak, aby używał funkcji, dopisz funkcję z sortowaniem szybkim (quicksort) i zastosuj ją w programie. Porównaj czas sortowania tablicy dwoma metodami.

WYKŁAD

5

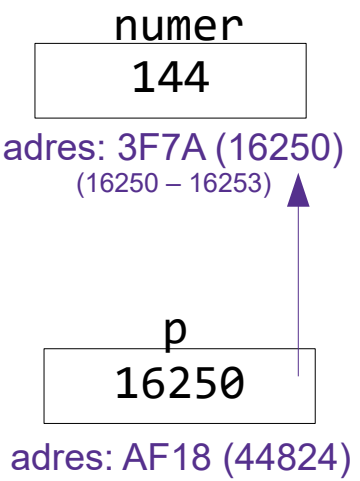
(10.12.2025)

WSKAŹNIKI. DYNAMICZNA ALOKACJA PAMIĘCI

Każda komórka (1 Bajt) w pamięci RAM posiada swój własny numer (adres) zachowany w formacie heksadecymalnym (HEX, szestnastkowym) Taki format zapisu jest krótszy niż binarny i dziesiętny.

Wskaźnik (ang. pointer) – zmienna, która przechowuje adres w pamięci RAM innej zmiennej (jej pierwszego Bajta).

Dynamiczna alokacja pamięci (na stercie, ang. heap) – w każdym momencie, na żądanie, nie tylko przy uruchomieniu i zakończeniu programu (statyczna alokacja pamięci, na stosie, ang. stack).



Format heksadecymalny zapisu liczb

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

3A5F

0x3A5F – z przedrostkiem 0x

3A5Fh – w asemblerze

3 = 3

A = 10

5 = 5

F = 15

$3 \cdot 16^3 + 10 \cdot 16^2 + 5 \cdot 16^1 + 15 \cdot 16^0 = 14943$

w binarnym:
11101001111111

Użycie wskaźnika pojedynczej zmiennej

```
int main()
{
   int numer=144;                    //zmienna int (4 bajty w RAM)
   int *w;              //typ zmiennej, *, nazwa wskaźnika w kodzie
   w=&numer;     //adres zmiennej numer przypisujemy na wskaźnik
   cout<<w<<endl;                          //wypisz wskaźnik (hex)
   cout<<*w<<endl;//wypisz zmienną, na którą wskazuje wskaźnik
   return 0;
}
```

& (ampersand) - operator adresu, umożliwia uzyskanie adresu w pamięci obiektu stojącego po jego prawej stronie

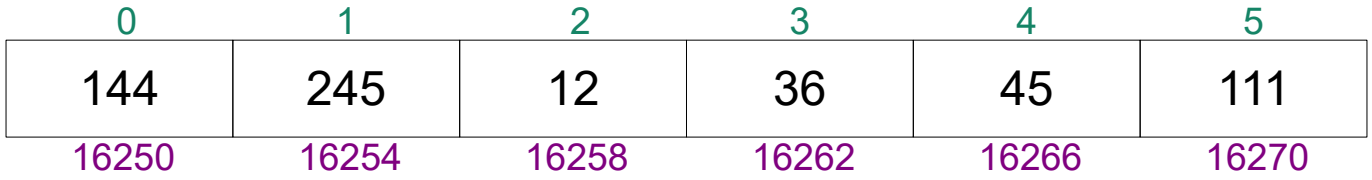
* - operator dereferencji, umożliwia dostęp do wartości, która znajduje się pod adresem wskazywanym przez wskaźnik po jego prawej stronie

Niektóre zalety stosowania wskaźników:

1) Bezpośredni dostęp do pamięci (adresów) co daje dużą kontrolę nad działaniem programu.
2) Dynamiczne zarządzanie pamięcią RAM, możliwość tworzenia i usuwania obiektów za pomocą operatorów `new` i `delete`.
3) Przekazywania dużych struktur do funkcji (tablic) bez konieczności ich kopiowania.
4) Zwiększenie szybkości zapisu/odczytu komórek w tablicy.
5) Możliwość współpracy z urządzeniami zewnętrznymi poprzez dostęp do odpowiednich adresów komórek pamięci.

Wskaźnik tablicy (dynamiczna alokacja pamięci)

```
int tab[6];  //alokacja statyczna
```




```
int rozm=10;                                    //rozmiar tablicy
int *tab;                          //deklaracja wskaźnika o nazwie tab
tab=new int[rozm];                          //dynamiczna alokacja tablicy
for(int i=0; i<rozm; i++)
{
  cout<<tab<<"   "<<*tab<<endl;      //adres komórki i jej zawartość
  tab++;          //inkrementacja wskaźnika (4 dla int, 8 dla double)
}
delete [] tab; //usuń tablicę o nazwie tab (zawsze jeżeli było new)
```

```
p=&tab[0] ⇔ p=tab
//nazwa tablicy alokowanej statycznie jest adresem jej zerowego elementu
```

Przekazywanie funkcjom oryginalnych zmiennych z głównego programu

```
float srednia(float &x, float &y, float &z)
//float srednia(float x, float y, float z)
{
    float s=(x+y+z)/3;
    x+=1000; y+=1000; z+=1000;
    return s;
}

int main()
{
    float a=1.2, b=2.3, c=3.1;
    cout<<”przed funkcja: ”<<a<<”; ”<<b<<”; ”<<c<<endl;
    cout<<”funkcja: ”<<srednia(a,b,c)<<endl;
    cout<<”po funkcji: ”<<a<<”; ”<<b<<”; ”<<c<<endl;
    return 0;
}
```

Funkcje zawsze pracują na oryginalnych tablicach

```
float srednia(float tab[], int n)
float srednia(float tab, int n)
float srednia(float *tab, int n)
```

Warto zastanowić się, dlaczego?

Zadanie 23

Napisz prosty program demonstrujący, że funkcje zawsze pracują na oryginalnych tablicach zarówno w przypadku alokacji statycznej jak i dynamicznej.
Przeanalizuj poprawność powyższych nagłówków funkcji w kontekście przekazywania do funkcji tablicy o rozmiarze n.

Pomiar czasu pracy programu (powtórzenie)

```
#include <time.h>
#include <cstdlib>

clock_t start, stop;                    //2 zmienne typu clock_t
float tsec;                  //zmienna do przechowywania czasu w sek.
start=clock();                          //numer cyklu procesora

    // operacje do pomiaru czasu

stop=clock();                           //numer cyklu
procesora
tsec=(float)(stop-start)/CLOCKS_PER_SEC;
       //liczba cykli rzutowana na float / liczba cykli na sekundę
cout<<"czas operacji [s]: "<<tsec<<endl;
```

Szybkość zapisu / odczytu

Zadanie 24

Rozbuduj program z Zadania 21 (sortowanie bąbelkowe). Wykonaj taką samą operację sortowania, ale tym razem z użyciem wskaźników i dynamicznej alokacji. Porównaj czas sortowania w obu przypadkach.

Obydwa sortowania zrealizuj za pomocą funkcji.

Sprawdź, czy tym razem można utworzyć tablicę większych rozmiarów.

14. Złożone typy danych (2) (struktura)

11.06.2025

Struktura to złożony typ danych grupujący logicznie powiązane ze sobą dane różnego typu w jednym obszarze pamięci. Składowe struktury – **pola** mają swoje unikatowe nazwy. Struktury pozwalają w przejrzysty sposób opisywać złożone obiekty. Przykładem struktury może być informacja o książce, której pola będą zawierały: imię i nazwisko autora (łańcuchy tekstowe), tytuł (łańcuch tekstowy), rok wydania (liczba całkowita), nazwa wydawnictwa (łancuch tekstowy), numer ISBN (liczba całkowita lub łańcuch tekstowy) itp.

Budowa struktury:

```
struct nazwa_typu_strukturalnego {
typ1 nazwa_składowej1;
typ2 nazwa_składowej2;
typ3 nazwa_składowej3;
.
.
.
};
```

Definicja struktury musi kończyć się średnikiem.

Przykładowa struktura może wyglądać następująco:

```
struct Book {
    string title;
    string aut_name;
    string aut_surname;
    int year;
    int isbn;
};
```

Tworzenie obiektu struktury

```
struct Osoba {
    string imie;
    string nazwisko;
    int r_ur;
} Kowalski, Nowak;
```

lub

```
Osoba Kowalski,Nowak;
```

(podobnie jak definicja zmiennych)

Dostęp do składowych struktury

```
nazwa_typu_strukturalnego.nazwa_składowej
```

```
Nowak.imie="Adam";
Nowak.nazwisko="Nowak";
Nowak.r_ur=1985;
cout<<Nowak.imie<<" "<<Nowak.nazwisko<<" "<<Nowak.r_ur<<endl;
```

Inicjowanie obiektów

(kolejność, przecinki, znak =)

w trakcie definicji struktury

```
struct Osoba {
    string imie;
    string nazwisko;
    int r_ur;
} Kowalski {"Jan","Kowalski",1964}, Nowak={"Adam","Nowak",1985};
```

lub po jej zdefiniowaniu:

```
Osoba Nowak={"Adam","Nowak",1985},Kowalski {"Jan","Kowalski",1964};
```

Tablica struktur

Tablica struktur umożliwia przechowywanie większej liczby obiektów danej struktury. Tablicę tworzymy podając jako jej typ nazwę struktury np.:

```
Osoba tab[10];
```

Powyższa tablica będzie przechowywała dane 10 osób. Możemy również utworzyć tablicę w sposób dynamiczny:

```
Osoba *tab2=new Osoba[10];
```

Korzystanie z takiej tablicy polega na podaniu jej nazwy, indeksu elementu w nawiasie kwadratowym, kropki i nazwy składowej:

```
tab[0].imie=”Adam”;
```

Przykład pokazujący jak wypełnić danymi całą tablicę:

```
for (int i=0;i<10;i++)
{
    tab[i].imie="Jan";
    tab[i].nazwisko="Kowalski";
    tab[i].r_ur=1980;
}
```

```
for (int i=0;i<10;i++)
{
    tab2[i].imie="Adam";
    tab2[i].nazwisko="Nowak";
    tab2[i].r_ur=1965;
}
```

Struktury – operator przypisania

Przypisanie struktur realizujemy za pomocą operatora przypisania:

```
Osoba Kowalski1 {"Jan","Kowalski",19},Kowalski2;
Kowalski2=Kowalski1;
```

W sytuacji, gdy wewnątrz struktury znajdują się wskaźniki, taka operacja przypisania jest dość niebezpieczna. Kompilator operację przypisania wykonuje przypisując wartości poszczególnych składowych jednej struktury do drugiej. Jeśli w strukturze występuje wskaźnik, to po takiej operacji wskaźniki w obu obiektach będą przechowywały takie same adresy. Tym samym po takiej operacji nie będziemy mieli dwóch w pełni niezależnych obiektów.

Obiekty w strukturze

Tworząc strukturę, jej składowe mogą być obiektami innych struktur. Liczba takich zagnieżdżeń nie jest w żaden sposób ograniczona. Taka sytuacja występuje w przykładzie obok →

Po utworzeniu obiektu chcąc dostać się do składowych struktury wewnętrznej musimy dwa razy skorzystać z operatora kropki →

```
struct DataUrodzenia {
    int r;
    int m;
    int d;
};

struct Student {
    string nazwisko;
    DataUrodzenia dataUr;
};

Student Jobs;
Jobs.dataUr.r=1955;
Jobs.nazwisko="Jobs";
```

W sytuacji gdy obiekt struktury jest składową innej struktury inicjowanie takiego obiektu danymi, wymaga podawania wartości w nawiasach klamrowych wewnątrz innych nawiasów klamrowych:

```
struct DataUrodzenia {

    int r;
    int m;
    int d;
};
struct Student {
    string nazwisko;
    DataUrodzenia dataUr;
};
Student kowalski={”Kowalski”,{2000,10,10}};
```

Jeśli podczas inicjowania obiektu w nawiasach klamrowych nie podamy żadnych wartości, to wszystkie składowe takiego obiektu będą wypełnione zerami.

Instrukcja:

```
Student nobody={};
```

Jest równoważna z instrukcjami:

```
nobody.nazwisko=””;
nobody.dataUr.r=0;
nobody.dataUr.m=0;
nobody.dataUr.d=0;
```

Wskaźnik na strukturę

Obiekty ze struktury mogą być tworzone z wykorzystaniem operatora `new`. W takim przypadku dostęp do składowych obiektu jest realizowany za pomocą operatora strzałki składającej się z dwóch znaków: `->`

Przykład tworzenia tablicy obiektów w sposób dynamiczny:

```
struct Osoba
{
    string imie;
    string nazwisko;
    int r_ur;
};

int main()
{
    Osoba *p,*p0;
    p=new Osoba[2];
    p0=p;
    p->imie="Adam";
    p->nazwisko="Nowak";
    p->r_ur=1986;
    p++;
    p->imie="Anna";
    p->nazwisko="Malinowska";
    p->r_ur=1977;
    for (int i=0;i<2;i++)
    {
       cout<<(p0+i)<<" "<<(p0+i)->imie<<" "<<(p0+i)->nazwisko<<" "<<(p0+i)->r_ur<<endl;
    }
    delete [] p;
    return 0;
}
```

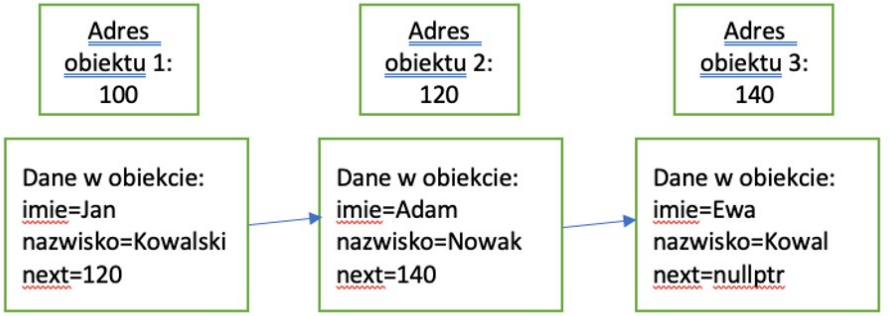
Lista jednokierunkowa

Tablice doskonale nadają się do przechowywania np. obiektów utworzonych ze struktury. Niestety tablica ma kilka wad. Dołożenie lub usunięcie elementu z jej środka jest bardzo wolną operacją, ponieważ zazwyczaj musimy przemieścić przy tym wiele innych elementów. Rozwiązaniem problemu jest w takiej sytuacji lista. Listę można opisać jako uszeregowany zbiór elementów. Każdy element zawiera dane oraz wskazuje na swojego następcę. Wskazuje, czyli przechowuje adres następnego elementu. Do tworzenia listy świetnie nadają się struktury. Poniżej znajduje się przykład struktury, która zostanie wykorzystana do utworzenia listy:

Wewnątrz struktury znajduje się wskaźnik na tę samą strukturę. Zadaniem tego wskaźnika będzie przechowywanie adresu kolejnej osoby. Jeśli to będzie ostatnia osoba to wskaźnik ten będzie równy `nullptr`.

```
struct Osoba
{
    string imie;
    string nazwisko;
    Osoba *next;
};
```

Do sprawnego posługiwania się listą potrzebujemy wskaźnika, który będzie przechowywał adres pierwszego obiektu naszej listy. Dodawanie nowego obiektu do listy polega na przydzieleniu pamięci na obiekt i zapisaniu jego adresu do wskaźnika **`next`** ostatniego obiektu listy. Wygląd listy trzech obiektów pokazuje schemat:



Usunięcie obiektu z listy wymaga zmiany wskaźnika `next` oraz usunięcia obiektu z pamięci. Jeśli usuwamy drugi obiekt, to najpierw musimy wykonać operację po której pierwszy obiekt będzie przechowywał adres trzeciego obiektu. Dopiero wtedy możemy usunąć z pamięci obiekt który był jako drugi. Poniżej pokazano przykład tworzenia listy oraz dodawania obiektów do listy:

```
struct Auto
{
    string marka;
    Auto *next=nullptr;
};

int main()
{
    Auto *pierwsze, *nowe, *ostatnie, *temp;

    nowe=new Auto;    //dodanie pierwszego obiektu
    nowe->marka="Audi";
    pierwsze=nowe;
    ostatnie=nowe;

    nowe=new Auto;   //dodanie kolejnego obiektu
    nowe->marka="Volkswagen";
    ostatnie->next=nowe;
    ostatnie=nowe;

    nowe=new Auto;   //dodanie kolejnego obiektu
    nowe->marka="Mercedes";
    ostatnie->next=nowe;
    ostatnie=nowe;

    temp=pierwsze;
    while(temp)   //wypisanie nazw z listy
    {
        cout<<temp->marka<<endl;
        temp=temp->next;
    }
    return 0;
}
```

Zadanie 25 (prosta baza danych)

Napisz program do utworzenia i zarządzania prostą bazą danych tekstowych i numerycznych (np. książki, przedmioty kolekcjonerskie) z wykorzystaniem listy jednokierunkowej. Program powinien posiadać opcje wypisywania rekordów, dodawania rekordów do bazy, usuwania niechcianych rekordów oraz modyfikowania istniejących rekordów.

*Dodatkowym atutem będzie odczytywanie bazy z pliku tekstowego i zapisywanie do pliku nowej, zmodyfikowanej bazy danych.

15. Złożone typy danych (3)
(unia i typ wyliczeniowy)

18.06.2025

Ćwiczenia programistyczne (3)

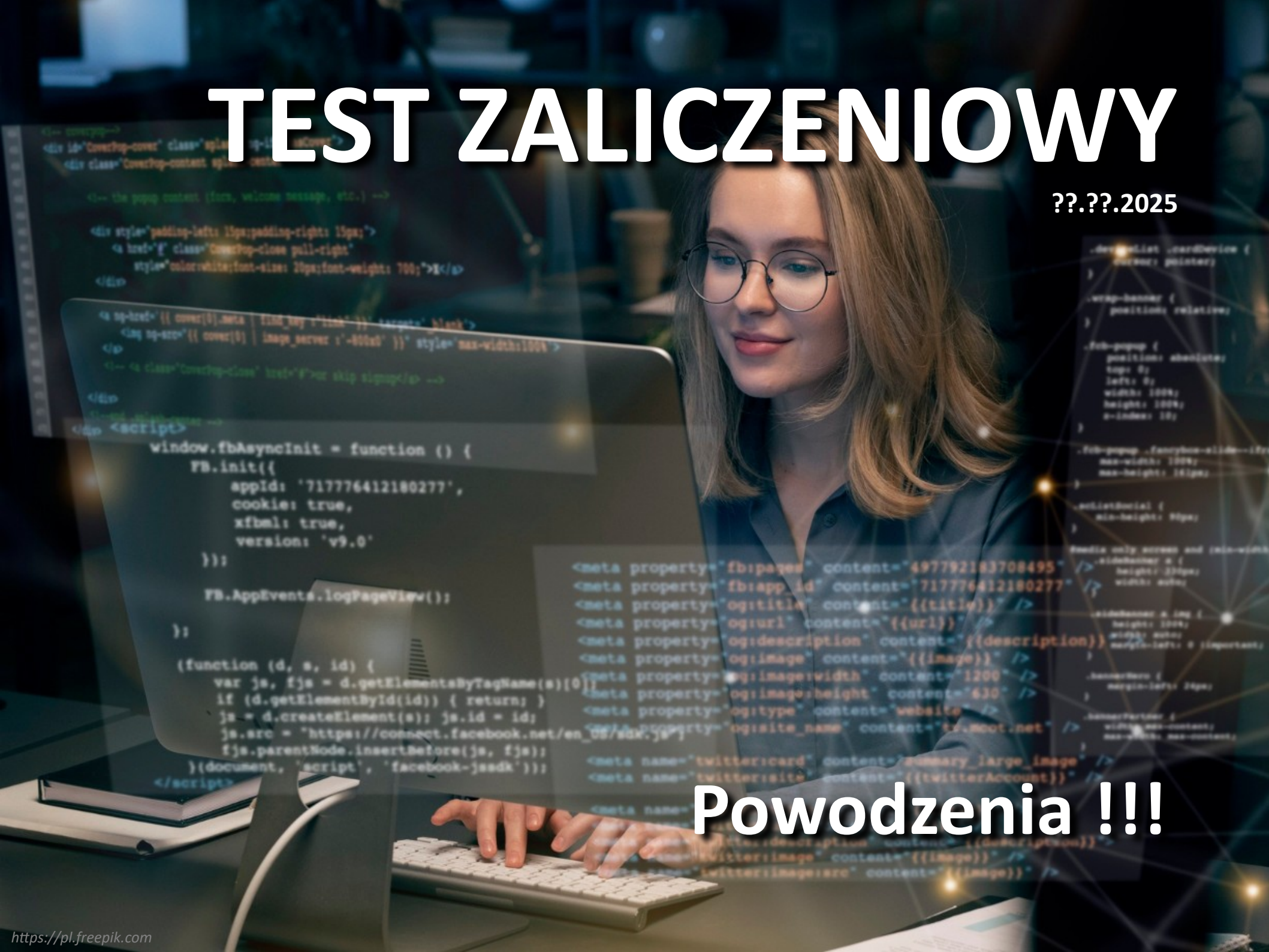
Zadanie 26 (analiza tekstu)

Napisz program, który wczyta cały tekst z pliku tekstowego, a następnie wykona analizę ilości znaków. Program wypisze ilość wszystkich znaków w tekście a następnie ilość konkretnych znaków, ale tylko i wyłącznie tych, które w tekście wystąpiły przynajmniej raz.

Ćwiczenia programistyczne (4)

Zadanie 27 (problem komiwojażera)

Napisz program, który rozwiąże problem komiwojażera. Zdefiniuj w pliku tekstowym (lub wylosuj) współrzędne miejsc dostarczenia przesyłek w wybranym kartezjańskim układzie współrzędnych. Program powinien znaleźć najkrótszą drogę pomiędzy punktami zaczynając od pierwszego z nich.


TEST ZALICZENIOWY
??.??.2025
Powodzenia !!!
https://pl.freepik.com