C++ składnia + biblioteki

> Kroki_Poznania – Taksonomia_Blooma https://bit.ly/3sD2XYA 1.Wiedza 2.Rozumienie 3.Zastosowanie 4.Analiza 5.Synteza 6.Ocenianie

Konfiguracja IDE

INSTALACJA – MinGW i Visual Studio Code
https://youtu.be/jla3qEnAFx0
https://nuwen.net/mingw.html
https://code.visualstudio.com/

Rozszerzenia plików

> C++ → plik `.cpp` / plik `.c`
> Pliki nagłówkowe → plik `.hpp` / plik `.h`

Uruchamianie programu

> `g++ program1.cpp -o p1` - Stworzenie pliku wykonującego `p1.exe`

> `g++ *.cpp -o p1` - Stworzenie pliku wykonującego `p1.exe` dla programu obiektowego, (wywołuje wszystkie pliki bieżącego folderu) – 1 folder (`src`) = jedna funkcja główna + pod funkcje i inne pliki

> `./p1` - Uruchomienie pliku `p1.exe`

Szablon podstawowy !!!

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

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

```cpp
OMÓWIENIE:
#include <iostream> // Biblioteka - udostępnia różnorakie funkcje programowi - łatwość
using namespace std; // Użycie standardowej przestrzeni nazw bo inaczej trzeba ciągle powtarzać std::cout itp.

int main(){} // Funkcja główna - obowiązkowa do działania

    cout << "Hello world!" <<endl; // Komunikaty w terminalu + zakończenie linii
    cout << "Hello world! \n"; // Komunikaty w terminalu + zakończenie linii
    cout << zmienna; // Wyprowadzenie zmiennej w terminalu
    return 0; // Funkcja zwraca 0 - nic
```

Wprowadzenie C ++

Komentarze

> Komentarze - notatka dla programisty - narzędzie debugujące
> Można za/odkomentować skryptem → Zaznaczenie → `Ctrl + /`

```cpp
// Komentarz jednoliniowy

/*
    Komentarz wielowierszowy
    Komentarz wielowierszowy
*/
```

Zmienne, literały i stałe

Zmienne

> **Zmienna** - kontener do przechowywania danych.
> Aby wskazać obszar przechowywania, każdej zmiennej należy **nadać niepowtarzalną nazwę** (identyfikator).
> Wartość zmiennej można zmienić, stąd nazwa **zmiennej** .

> **Zasady nazywania zmiennej**
>
> - Nazwa zmiennej może zawierać tylko litery, cyfry i podkreślenie `_` .
> - Nazwa zmiennej nie może zaczynać się od liczby.
> - Nazwy zmiennych nie powinny zaczynać się wielką literą.
> - Nazwa zmiennej nie może być słowem kluczowym . Na przykład `int` to słowo kluczowe używane do oznaczania liczb całkowitych.
> - Nazwa zmiennej może zaczynać się od podkreślenia. Jednak nie jest to uważane za dobrą praktykę.

Literały języka

> **Literały** to dane używane do przedstawiania stałych wartości. Można ich używać bezpośrednio w kodzie. Np.: `1` , `2.5` , `'c'` itd.

> 1. Liczby całkowite `1` - w różnych układach liczbowych
> 2. Liczby zmiennoprzecinkowe `2.5` - w różnych układach liczbowych
> 3. Znaki `F` `}` `a` itd.
>
> ```
> \b  - Backspace - usunięcie ostatniego znaku
> \f  - Form feed - wysuw strony - ?Linux? - przenosi aktywną pozycję do
> pozycji początkowej na początku następnej strony logicznej - cokolwiek
> to znaczy
> \n  - Nowa linia
> \r  - Powrót do początku linii
> \t  - Tabulator
> ```

```
\v  - Zakładka pionowa - ?Linux? -niedziałan na wszystkich terminalach
\\  - Ukośnik wsteczny
\'  - Pojedynczy cudzysłów
\"  - Podwójny cudzysłów
\?  - Znak zapytania
\0  - Znak zerowy
```

4. Sekwencje ucieczki `\n`
5. Łańcuch znaków

```
"good"  - stała łańcuchowa
""      - stała łańcuchowa zerowa
" "     - stała łańcuchowa sześciu białych znaków
"x"     - stała łańcuchowa o pojedynczym znaku
"Earth is round\n"  - wypisuje łańcuch z nową linią
```

Stałe

W C ++ możemy tworzyć zmienne, których wartości nie można zmienić. W tym celu używamy `const` słowa kluczowego, np.:

```
const int LIGHT_SPEED = 299792458;
LIGHT_SPEED = 2500 // Error! LIGHT_SPEED is a constant.
```

Stałą można również utworzyć za pomocą `#define` dyrektywy preprocesora - `C++ Macros.`

Typy zmiennych / danych

Wielkość typu może różnić się zależnie od systemu operacyjnego?!
Modyfikatory typu - zmieniają przedział liczbowy określonego typu:

- `signed` - zmienia przedział typu na liczby dodatnie i ujemne: -1, 0, 1 - domyślnie

- `unsigned` - zmienia przedział typu tylko na liczby dodatnie: 0, 1
- `short` - typ ma mniejszy rozmiar
- `long` - typ ma większy rozmiar

Typy całkowite - przechowuje liczby całkowite np.: -1 , 0, 1

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`short`	2	[-32768, 32767]	`short x = -1;`
`int`	4	[-2147483648, 2147483647]	`int x = -11;`
`long`	4	[-2147483648, 2147483647]	`long x = -111;`
`long long`	8	[-9223372036854775808, 9223372036854775807]	`long long x = -1111;`
`unsigned short`	2	[0, 65535]	`unsigned short x = 1;`
`unsigned int`	4	[0, 4294967295]	`unsigned int x = 11;`
`unsigned long`	4	[0, 4294967295]	`unsigned long x = 111;`
`unsigned long long`	8	[0, 18446744073709551615]	`unsigned long long x = 1111;`

Typy rzeczywiste / zmiennoprzecinkowe - przechowuje liczby rzeczywiste np.: -64.74, 0, 134.64534

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`float`	2	pojedyncza precyzja - dokładność 6 - 7 cyfr po przecinku	`float` x = 1.11;
`double`	8	podwójna precyzja - dokładność 15 - 16 cyfr po przecinku	`double` x = 1.1111;
`long double`	12	liczby z ogromną dokładnością - 19 - 20 cyfr po przecinku	`long double` x = 1.111111;

> **Typy znakowe** - przechowuje znaki z klawiatury - oprócz poniższy istnieją jeszcze: `har8_t`, `char16_t`, `char32_t` → cholera wie po co

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`char`	1	[-128, 127]	`char` x = 'h';
`unsigned char`	1	[0, 255]	`unsigned char` x = 'h';
`wchar_t`	2	szeroki char	`wchar_t` x = L'␣';

> **Typy łańcuchowe** - łańcuch znaków - napisy - wielkość skalowalna

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`string`	*	*	`string` x = "Ciąg znaków?!"

> **Typy logiczne** - używane w instrukcjach warunkowych i pętlach - wartości TAK lub NIE

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`bool`	1	[true (1) lub false (0)	`bool` x = false;

> **Typy puste** - wskazuje na brak danych - oznacza „nic” lub „brak wartości”

Typ danych / zmiennych	Rozmiar w bajtach	Zakres	Przykład zastosowania
`void`	0	-	Używany w funkcjach i wskaźnikach

> **Pochodne typy danych** - typy danych, które pochodzą z podstawowych typów danych, są typami pochodnymi. Na przykład: tablice, wskaźniki, typy funkcji, struktury itp.

Podstawowe operecje I/O

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

int main()
{
    int liczba1, liczba2;
    cout << "Podaj liczbe: \n";
    cin >> liczba1;
    cin >> liczba2; // Wprowadzenie zmiennych
    cout << "Liczba1 wynosi: " << liczba1 << endl;
```

```
    /*lub*/ cin >> liczba1 >> liczba2; // Wprowadzenie wielu zmiennych
    cout << liczba1 << " i " << liczba2 << " to liczby \n";
    return 0;
}
```

Konwersja typu

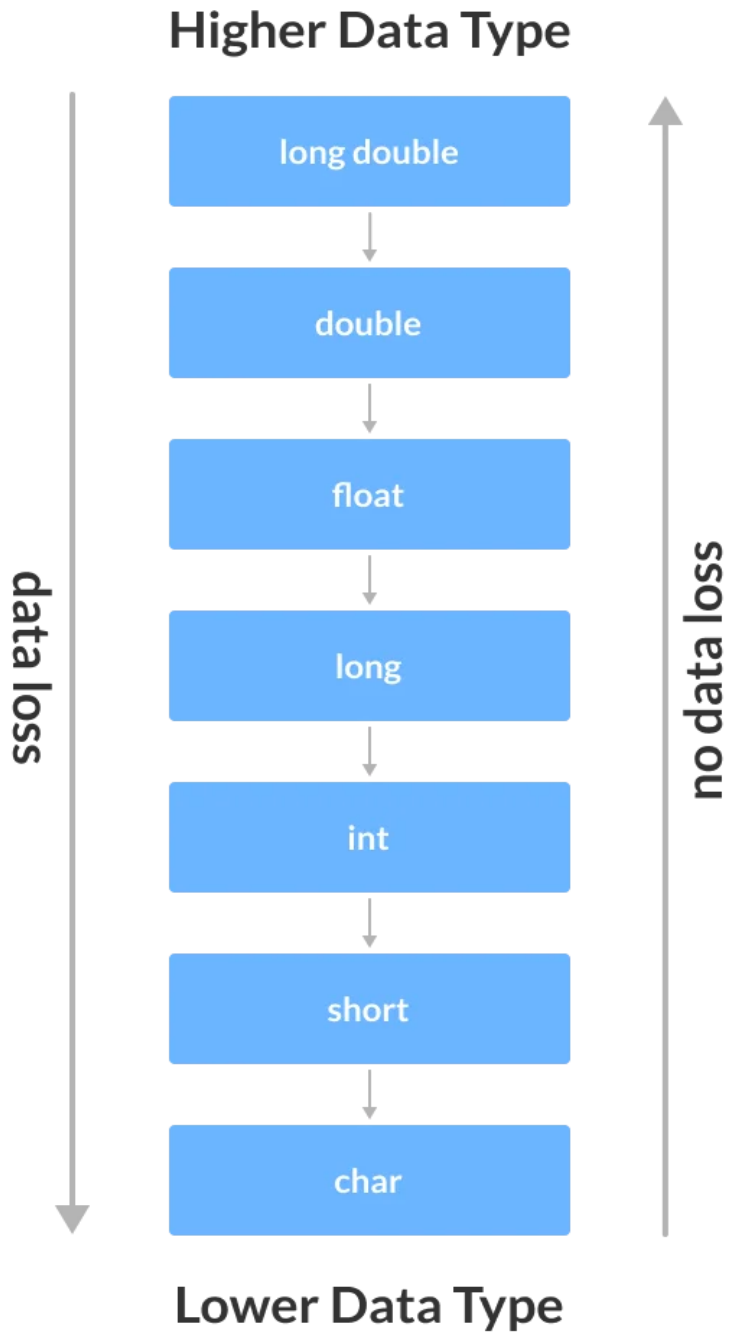
Niejawna

> Konwersja typu, która jest wykonywana automatycznie przez kompilator, jest nazywana **niejawną konwersją typu**. Ten typ konwersji jest również nazywany **konwersją automatyczną.**

```
    // Przypisanie liczby całkowitej na zmiennoprzecinkową
    int num_int = 9;
    double num_double;
    num_double = num_int;

    cout << "num_int = " << num_int << endl;
    cout << "num_double = " << num_double << endl;

        Wynik:
            num_int = 9
            num_double = 9
    /* lub */

    // Przypisanie liczby zmiennoprzecinkowej na całkowitą
    int num_int;
    double num_double = 9.99;
    num_int = num_double;

    cout << "num_int = " << num_int << endl;
    cout << "num_double = " << num_double << endl;

        Wynik:
            num_int = 9
            num_double = 9.99
```

Utrata danych

Konwersja z jednego typu danych na inny jest podatna na **utratę danych**. Dzieje się tak, gdy dane większego typu są konwertowane na dane mniejszego typu.



Jawna

Odlewanie typu - notacja odlewana
(typ_danych)wyrażenie;

```
int num_int;
double num_double = 3.56;

num_int = (int)num_double; // konwersja z int na double
cout << "num_int = " << num_int << endl;
```

Odlewanie stylu funkcji - Rzutowanie typów
data_type(wyrażenie);

```
int num_int;
double num_double = 3.56;

num_int = int(num_double); // konwersja z int na double
cout << "num_int = " << num_int << endl;
```

Przypisanie

```
double num_double = 3.56;
cout << "num_double = " << num_double << endl;

int num_int1 = (int)num_double; // przypisanie odlewania typu
cout << "num_int1 = " << num_int1 << endl;

int num_int2 = int(num_double); // przypisanie odlewania stylu funkcji
cout << "num_int2 = " << num_int2 << endl;
```

Operatory konwersji typów - później:
`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`

Operatory

```
DZIAŁANIA i OPERATORY:
Operatory arytmetyczne:
    +   Dodanie          2 + 2 = 4
    -   Odejmowanie      5 - 2 = 3
    *   Mnożenie         3 * 3 = 9
    /   Dzielenie        22 / 8 = 2.75 // int 2, float 2.75
    %   Moduł / Reszta z dzielenie  22 % 8 = 6

Operatory przypisania wartości:
    =   Przypisanie wartości z lewej do prawej; znaczenie:  i=1;
    +=  Dodawanie; znaczenie:   x = x + y;
    -=  Odejmowanie; znaczenie: x = x - y;
    *=  Mnożenie; znaczenie:    x = x * y;
    /=  Dzielenie; znaczenie:   x = x / y;
    %=  Moduł / Reszta z dzielenie  x = x % y;

Operatory inkrementacji i dekrementacji:
    ++  Inkrementacja; znaczenie: i=i+1; // i++ lub ++i
    --  Dekrementacja; znaczenie: i=i+1; // i-- lub --i

Operatory porównania / relacji:
    !   Zaprzeczenie; znaczenie:    x ! y;
    ==  Równy; znaczenie:           x == y;
    !=  Różny; znaczenie:           x != y;
    <   Mniejszy; znaczenie:        x < y;
    >   Większy; znaczenie:         x > y;
    <=  Mniejszy równy; znaczenie:  x <= y;
    >=  Większy równy; znaczenie:   x >= y;

Operatory logiczne - działają po kolei do lewej - zwracają bool -> T/F:
    !   Zaprzeczenie / Negacja  NOT ("nie"); znaczenie: x ! y;
    &&  Koniunkcja  AND ("i"); znaczenie:       x && y;
    ||  Alternatywa OR  ("lub"); znaczenie:     x || y;

    bool result;
    result = (3 != 5) && (3 < 5);     // true

Operatory bitowe
    &   Binarne AND
    |   Binarny OR
```

```
    ^   Binarny XOR
    ~   Uzupełnienie binarnego One
    <<  Binarne Shift w lewo
    >>  Binary Shift w prawo

Inne operatory
    sizeof  zwraca rozmiar typu danych  sizeof(int); // 4

    ?:      zwraca wartość na podstawie warunku string result = (5 > 0) ?
"even" : "odd"; // "even"

    &       reprezentuje adres pamięci operandu # // address of num

    .       uzyskuje dostęp do elementów składowych zmiennych struktur lub
obiektów klas    s1.marks = 92;

    ->      używane ze wskaźnikami w celu uzyskania dostępu do zmiennych klasy
lub struktury    ptr->marks = 92;

    <<      wypisuje wartość wyjściową  cout << 5;
    >>      pobiera wartość wejściową   cin >> num;
```

Kontrola przepływu w C ++

If ... else

> **Funkcja warunkowa** `if`, `if ... else`, `if ... else if ... else` i oraz zwgnieżdżony if
>
> - Od spełnienia warunków zależy który blok kodu zastanie wykonany
> - Do warunków logicznych używa się operatorów
> - Instrukcje można **zagnieżdżać** (umieszczać jedna w drugiej)
> - W wypadku przedziałów zaczynamy do **jednego z ekstremów** i przesuwa się spełniając kolejne warunki np.: przedział wiekowe

> `if` - funkcja prosta - gdy spełniony jest warunek, wykona się blok kodu

```
if ( warunek_logiczny )
{
    /* code */
}
```

`if ... else` - w zależności czy warunek jest spełniony wykonają się różne bloki kodu

```
if ( warunek_logiczny )
{
    /* code_1 */
}
else
{
    /* code_2 */
}
```

`if ... else if ( ... else)` - gdy else nie wystarczy i potrzebujemy sprawdzić kolejny warunek

```
if ( warunek_logiczny_1 )
{
    /* code1 */
}
else if ( warunek_logiczny_2 )
{
    /* code_2 */
}
else
{
    /* code_3 */
}
```

> **Zagnieżdżenie** + PRZYKŁAD

```
    if (wiek > 0)
    {   // Gdy blok kodu ma więcej niż 1 linijkę kodu { kod w nawiasach }
        // Funkcja zagnieżdżona
        if (wiek < 20) // Operatory logiczne i matematyczne
            cout << "młody"; // Gdy blok kodu ma 1 linijke - bez nawiasów {}
        else if (wiek < 60)
            cout << "dorosły";
        else if (wiek < 120)
            cout << "stary";
        else
            cout << "Ludzie tyle nie żyją";
    }
    else
        cout << "Ludzie tyle nie żyją";
```

Pętla for

> **Pętle**
>
> - Stosowane przy powtarzaniu bloku kodu określoną liczbe razy
>   - Inicjalizacja - inicjalizuje zmienną - wykonywane tylko raz
>   - Warunek - spełniony = wykonanie, niespełniony = zakończenie
>   - Aktualizacja - aktualizuje wartość zainicjalizowanych zmiennych i ponownie sprawdza ich stan

```
    for( inicjalizacja; warunek; aktualizacj ) // warunek for (;;) - pętla nieskończona
    {
        /* code */
    }

PRZYKŁAD_1:
    for (int i = 0; i <= 5; i++) // lub ++i <- XD
```

```
    {
        cout << i;
    }

PRZYKŁAD_2 - pętla zagnieżdżona:
    int x;
    cin>>x;

    for (int i = 1; i <= x; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            if (i==x || j == 1 || j == i)
            {
                cout<<"*";
            }
            else
            {
                cout<<" ";
            }
        }
        cout<<endl;
    }
Wynik:
4
*
**
* *
****
```

Pętla while

```
    while( warunek ) // warunek = while( true ) - pętla nieskończona
    {
        instrukcja;
        aktualizacja;
    }

PRZYKŁAD:
    int i = 0;
```

```
    while (i <= 5)
    {
        cout << i;
        i++;
    }
```

Pętla do ... while

```
    do
    {
        /* code */
        aktualizacja;
    } while ( warunek );

PRZYKŁAD:
    int i = 0;
    do
    {
        cout << i;
        i++;
    } while(i <= 5); // int = 0; i warunek = while( i = 0 ) - pętla
nieskończona
```

Break

> Instrukcja `break` kończy pętlę, gdy zostanie napotkana, występuje w funkcji wielokrotnego wyboru `switch`

```
    break;
```

```
PRZYKŁAD_1:
    for (int i = 1; i <= 5; i++)
    {
        if (i == 3)
        {
            break;
```

```
        }
        cout << i << endl;
    }

Wynik:
1
2
```

```
PRZYKŁAD_2:
    while (true)
    {
        cout << "Wczytaj liczbe: ";
        cin >> number;

        if (number < 0)
        {
            break;
        }

        sum += number;
    }
    cout << "Suma wynosi " << sum << endl;

Wynik:
Wpisz liczbe: 1
Wpisz liczbe: 2
Wpisz liczbe: 3
Wpisz liczbe: -5
Suma wynosi 6
```

```
PRZYKŁAD_3:
    int number;
    int sum = 0;

    for (int i = 1; i <= 3; i++)
    {
        for (int j = 1; j <= 3; j++)
        {
            if (i == 2)
```

```
            {
                break;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

Wynik:
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

Continue

> Instrukcja `continue` służy do pominięcia bieżącej iteracji pętli, a sterowanie programem przechodzi do następnej iteracji.

```
    continue;
```

```
PRZYKŁAD_1:
    for (int i = 1; i <= 5; i++)
    {
        if (i == 3)
        {
            continue;
        }

        cout << i << endl;
    }

Wynik:
1
2
```

```
4
5
```

```
PRZYKŁAD_2:
    int sum = 0;
    int number = 0;

    while (number >= 0)
    {
        sum += number;

        cout << "Wpisz liczbe: ";
        cin >> number;

        if (number > 50)
        {
            cout << "Liczba jest większa niż 50 i nie zostanie obliczona." <<
endl;
            number = 0;
            continue;
        }
    }
    cout << "Suma wynosi " << sum << endl;

Wynik:
Wpisz liczbe: 12
Wpisz liczbe: 0
Wpisz liczbe: 2
Wpisz liczbe: 30
Wpisz liczbe: 50
Wpisz liczbe: 56
Liczba jest większa niż 50 i nie zostanie obliczona.
Wpisz liczbe: 5
Wpisz liczbe: -3
Suma wynosi 99
```

```
PRZYKŁAD_3:
    int number;
```

```
    int sum = 0;

    for (int i = 1; i <= 3; i++)
    {
        for (int j = 1; j <= 3; j++)
        {
            if (j == 2)
            {
                continue;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

Wynik:
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

Switch ... case i enum

> `switch` - funkcja wielokrotnego wyboru
>
> - Pozwala dokonać jednego wyboru, co do wykonania pewnego bloku kodu (wielokrotnie w nieskończonej pętli)
> - Warto zamieniać go z if ... else if, gdy warunki wyglądają tak: `x == 1, x == 2, x == 3` , itd.

```
    switch ( zmienna )
    {
    case wartość_1: // char -> 'x':    int -> 1: // Przypadek 1
        /* code_1 */
        break;

    case wartość_2: // Przypadek 2
```

```
        /* code_2 */
        break;

    case wartość_n: // Przypadek n
        /* code_n */
        break;

    default: // Wartość domyślna, wykonywana, gdy wprowadzona zmienna nie
zawiera się w poprzednich wyborach
        /* code_domuślny */
        break;
    }

PRZYKŁAD:
    switch (x)
    {
    case 1:
        {   // Gdy blok kodu ma więcej niż 1 linijka kodu { kod }
            cout << "To jest 1 \n";
            cout << "To na pewno jest 1 \n";
            break;
        }

    case 2:
        cout << "To jest 2 \n"; // Gdy blok kodu ma 1 linijkę - bez nawiasów {}
        break;

    default:
        cout << "Nie ma takiej opcji do wyboru \n";
        break;
    }
```

> `enum` - typ wyliczeniowy
>
> - pozwala na zadeklarowanie stałych liczbowych
> - współgra z switch
> - przed funkcją main -numeracja zaczyna się od 0 - można przypisać inną

```
    enum nazwa_wyliczenia
    {
        element_0,      // 0 // Inna nazwa - element_0 dla elementu o nr 0
        element_1,      // 1
        element_2 = 3, // 3 -> Przypisanie innej wartości elementowi
        element_n       // 4
    };

int main()
{
    int x;
    cin >> x;
    switch (x)
    {
    case element_0:
        /* code_1 */ cout << "LOL";
        break;
    case element_1:
        /* code_2 */
        break;
    case element_2:
        /* code_3 */
        break;
    case element_n:
        /* code_n */
        break;

    default:
        /* code_domuślny */
        break;
    }
    return 0;
}

WYJAŚNIENIE: Teraz do wyboru switch można użyć zarówno int jaki i nazwy stałej
liczbowej ktorej można nadać dowolną nazwę i która jest odzwierciedleniem
liczby int np.: nazwy kolorów

Przykładowy wynik:
element_0
```

```
LOL
    // lub
0
LOL
```

Goto

> Instrukcja `goto` służy do zmiany normalnej kolejności wykonywania programu poprzez przekazanie kontroli do innej części programu
>
> - Należy unikać instrukcji przejścia `goto` , ponieważ sprawia, że logika programu jest złożona i zagmatwana

```
goto nazwa_etykiety; // Przejście do etykiety, pominięcie bloków kodu po goto
nazwa_etykiety;
    kod ...
nazwa_etukiety: // Po przejściu do etykiety wykonywany są bloki kodu poniżej
    inny blok kodu ...

PRZYKŁAD:
    float num, srednia, sum = 0.0;
    int i, n;

    cout << "Maksymalna liczba wejsc: ";
    cin >> n;

    for (i = 1; i <= n; ++i)
    {
        cout << "Wpisz n" << i << ": ";
        cin >> num;

        if (num < 0.0)
        {
            goto jump; // Sterowanie programem przejdź do jump:
        }
        sum += num;
    }
```

```
jump:
    srednia = sum / (i - 1);
    cout << "\nSrednia = " << srednia;

Wynik:
Maksymalna liczba wejsc: 10
Wpisz n1: 2.3
Wpisz n2: 5.6
Wpisz n3: -5,6
Średnia = 3,95
```

Funkcje C++

Funkcje

> **Funkcja** to blok kodu o nadanej nazwie, który wykonuje określone zadanie
>
> - Polega na **rozłożeniu problemu** na mniejsze części
> - Powoduje, że cały program jest **łatwy** do zrozumienia
> - Napisany blok kodu **można ponownie używać**

> **Własności funkcji:**
>
> - Posiada własną nazwę - **Nazwa** powinna określać **zadanie funkcji, czasownik, bez spacji**
> - Może posiadać **listę argumentów** potrzebną do działania funkcji
> - Może **zwracać wartość**

> Istnieją **dwa rodzaje funkcji**:
>
> - **Standardowe funkcje biblioteczne**: wstępnie zdefiniowane w C ++
> - **Funkcja zdefiniowana** / utworzona przez użytkownika

Deklaracja funkcji

> **Deklaracja funkcji** - ciało funkcji wykonujące operacje

```
typ_zwracanej_zmiennej nazwa_funkcji( typ_arg1 arg1, typ_arg2 arg2, ... )
    {
        // code
    }

PRZYKŁAD:
void witaj_swiecie()
{
    cout << "Hello World";
}
```

Wywołanie funkcji

> **Wywołanie funkcji** - po nazwie do pracy (jak koleżankę na studiach - po imieniu)

```
int main()
{
    nazwa_funkcji(arg1, arg2);
}

PRZYKŁAD - Wywołanie napisu :
    // Deklaracja funkcji
    void
    witaj_swiecie()
{
    cout << "Hello World";
}
// Wywoływanie funkcji
int main()
{
    witaj_swiecie();
    return 0;
}
```

Parametry / argumenty funkcji

> **Parametry / argumenty funkcji** - deklaracja typu skopiowanych danych i nadanie im nowej nazwy używanej w danej funkcji - więcej o tym później

```
PRZYKŁAD_1:

void printNum(int num) // Kopia wartości, nadanie jej innej nazwy i operacja na niej
{
    cout << num;
}

int main()
{
    int n = 7;
    printNum(n); // Wysłanie do funkcji wartości do pracy
    return 0;
}

PRZYKŁAD_2:
void displayNum(int n1, float n2) // Deklaracja kilku parametrów
{
    cout << "Liczba int to " << n1;
    cout << "Liczna double to " << n2;
}

int main()
{
    int num1 = 5;
    double num2 = 5.5;

    displayNum(num1, num2); // Wysłanie kilku wartości - odczytywane są one pokolei

    return 0;
}
```

Return - instrukcja zwrotu

> `return x;` - zwraca wartość obliczoną przez funkcje, typ wartości równa się typ_zwracanej_zmiennej na początku deklarowania funkcji

```
PRZYKŁAD:
int add(int a, int b)
{
    return (a + b); Instrukcja zwrotu - zwraca sume argumentów
}

int main()
{
    int sum;
    sum = add(100, 78); // Zwraca sume 100 + 78, czyli int 178, który zostaje przypisany do zmiennej sum

    cout << "100 + 78 = " << sum << endl;

    return 0;
}
```

Prototyp funkcji

> **Prototyp funkcji** - deklarowanie istnienia ciała funkcji w dalszej części programu
>
> - Zwiększa czytelność - gdy deklarowana funkcja jest długa
> - Niezbędny przy deklaracji funkcji - po funkcji głównej (main)

```
void add(int, int); // Prototyp funkcji

int main()
{
    add(5, 3); // Wywoływanie funkcji
    return 0;
}

void add(int a, int b) // Deklaracja funkcji
{
```

```
    cout << (a + b);
}
```

Funkcje biblioteki

> Funkcje biblioteczne to wbudowane funkcje w językach programowania
>
> - Programiści mogą używać funkcji bibliotecznych, wywołując je bezpośrednio, nie muszą samodzielnie pisać funkcji.
> - Aby korzystać z funkcji bibliotecznych, zwykle musimy dołączyć plik nagłówkowy, w którym te funkcje biblioteczne są zdefiniowane
> - Na przykład, aby używać funkcji matematycznych, takich jak `sqrt()` i `abs()`, musimy dołączyć plik nagłówkowy `cmath` - więcej w Biblioteki w C++

```
#include <iostream>
#include <cmath> // Dodanie biblioteki matematycznej
using namespace std;

int main()
{
    double num, pierwiastek;

    num = 25.0;
    pierwiastek = sqrt(num); // Wywołanie funkcji

    cout << "Pierwiastek kwadratowy z " << num << " = " << pierwiastek;

    return 0;
}
```

Typy funkcji

> Wszystkie poniższe programy są poprawne i dają ten sam wynik
>
> - Nie ma zasad, którą należy wybrać - wszystko zależy od sytuacji i sposobu rozwiązania problemu :D

Funkcja bez argumentu i bez wartości zwracanej

> Cały program zawiera się w funkcji, main tylko wywołuje

```
void prime()
{
    int num, i, flag = 0;

    cout << "Wpisz dodatnia liczbe calkowita, ktora chcesz sprawdzic: ";
    cin >> num;

    for (i = 2; i <= num / 2; ++i)
    {
        if (num % i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << num << " nie jest liczba pierwsza.";
    }
    else
    {
        cout << num << " jest liczba pierwsza.";
    }
}

int main()
{
    prime();
    return 0;
}
```

Funkcja bez argumentu, ale zwracająca wartość

>

> Funkcja main wywołuje funkcje w celu pobrania wartości

```
int prime();

int main()
{
    int num, i, flag = 0;


    num = prime(); // Żaden argument nie jest przekazywany do prime ()
    for (i = 2; i <= num / 2; ++i)
    {
        if (num % i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << num << " nie jest liczba pierwsza.";
    }
    else
    {
        cout << num << " jest liczba pierwsza.";
    }
    return 0;
}

int prime() // Zwracany typ funkcji to int
{
    int n;

    printf("Wpisz dodatnia liczbe calkowita, ktora chcesz sprawdzic: ");
    cin >> n;

    return n;
}
```

Funkcja z argumentem, ale bez wartości zwracanej

> Funkcja jest funkcją obliczeniową dającą pełną odpowiedź dla zapytania funkcji main

```
void prime(int n);

int main()
{
    int num;
    cout << "Wpisz dodatnia liczbe calkowita, ktora chcesz sprawdzic: ";
    cin >> num;

    prime(num); // Argument num jest przekazywany do funkcji prime ()
    return 0;
}

// Nie ma wartości zwracanej do wywołania funkcji. W związku z tym zwracany typ funkcji jest nieważny
void prime(int n)
{
    int i, flag = 0;
    for (i = 2; i <= n / 2; ++i)
    {
        if (n % i == 0)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
    {
        cout << n << " nie jest liczba pierwsza.";
    }
    else
    {
        cout << n << " jest liczba pierwsza.";
```

```
    }
}
```

Funkcja z argumentem i wartością zwracaną

> Funkcja jest funkcją obliczeniową zwracającą wartość dla funkcji main

```
int prime(int n);

int main()
{
    int num, flag = 0;
    cout << "Wpisz dodatnia liczbe calkowita, ktora chcesz sprawdzic: ";
    cin >> num;

    flag = prime(num); // Numer argumentu jest przekazywany do funkcji check ()

    if (flag == 1)
        cout << num << " nie jest liczba pierwsza.";
    else
        cout << num << " jest liczba pierwsza.";
    return 0;
}

int prime(int n) // Ta funkcja zwraca wartość całkowitą
{
    int i;
    for (i = 2; i <= n / 2; ++i)
    {
        if (n % i == 0)
            return 1;
    }

    return 0;
}
```

Przeciążenie funkcji

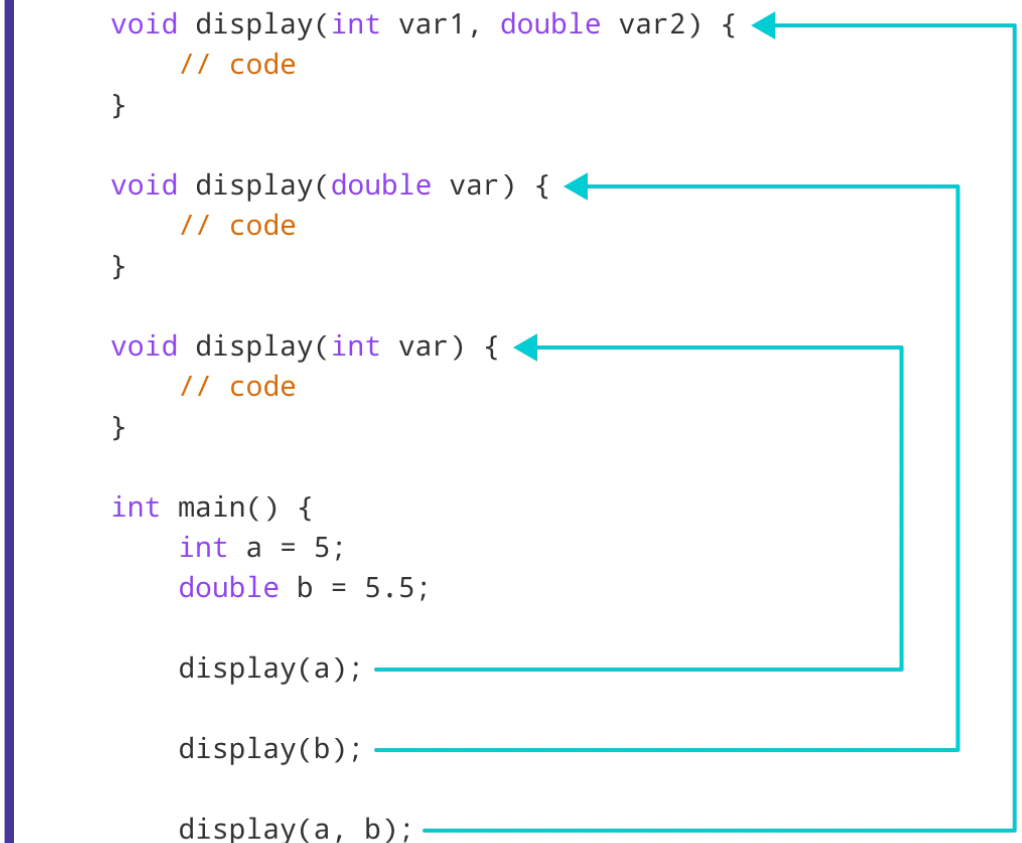
-

> **Przeciążenie funkcji** - funkcje mogą mieć **tę samą nazwę**, jeśli **liczba i / lub typ** przekazywanych argumentów **jest różny**

```
Poprawne:
int test() {}
int test(int a) {}
float test(double a) {}
int test(int a, double b) {}

Błędne:
int test(int a) {}
double test(int b){}

PRZYKŁAD:
void display(int var1, double var2) // Funkcja z 2 parametrami
{
    cout << "Liczba int: " << var1;
    cout << " i liczba double: " << var2 << endl;
}

void display(double var) // Funkcja z pojedynczym parametrem typu double
{
    cout << "Liczba double: " << var << endl;
}

void display(int var) // Funkcja z pojedynczym parametrem typu int
{
    cout << "Liczba int: " << var << endl;
}

int main()
{
    int a = 5;
    double b = 5.5;

    display(a); // Wywołanie funkcji z parametrem typu int
    display(b); // wywołanie funkcji z parametrem typu double
    display(a, b); // Wywołanie funkcji z 2 parametrami
```

```
    return 0;
}
```

```
void display(int var1, double var2) {
    // code
}

void display(double var) {
    // code
}

void display(int var) {
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);

    display(b);

    display(a, b);

    ... ...

}
```



Argument domyślny

- Jeśli wywoływana jest funkcja **z domyślnymi argumentami bez przekazywania argumentów**, używane są **domyślne parametry**
- Jeśli jednak **argumenty są przekazywane** podczas wywoływania funkcji, **domyślne argumenty są ignorowane**

Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp();
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 3 : All arguments are passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6, -2.3);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 4 : Second argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(3.4);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Error

- Gdy podamy domyślną wartość parametru, wszystkie kolejne parametry również muszą mieć wartości domyślne

```
// Niepoprawny
void add(int a, int b = 3, int c, int d);

// Niepoprawny
void add(int a, int b = 3, int c, int d = 4);

// Poprawny
void add(int a, int c, int b = 3, int d = 4);
```

- Definiowanie w prototypie

```
void display(char = '*', int = 3); // Lub (char c = '*', int count = 3)

int main()
{
    int count = 5;

    cout << "Nie przekazano argumentu: "; // 3 razy *
    display();

    cout << "Przekazano pierwszy argument: "; // 3 razy #
    display('#');

    cout << "Oba argumenty zostaly przekazane: "; // 5 razy $
    display('$', count);

    return 0;
}

void display(char c, int count)
{
    for (int i = 1; i <= count; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

- Jeśli definiujemy argumenty w deklaracji to owa deklaracja funkcji musi znajdować się przed wywołaniem tej funkcji np.: przed main

```
void display(char c = '*', int count = 3)
{
    for (int i = 1; i <= count; ++i)
    {
```

```
        cout << c;
    }
    cout << endl;
}

int main()
{
    int count = 5;

    cout << "Nie przekazano argumentu: "; // 3 razy *
    display();

    cout << "Przekazano pierwszy argument: "; // 3 razy #
    display('#');

    cout << "Oba argumenty zostaly przekazane: "; // 5 razy $
    display('$', count);

    return 0;
}
```

Klasa pamięci

> Zmienne w C++ mają 2 cech: **typ i klasę pamięci**
>
> - **Typ** przechowywanych danych
> - **Klasa pamięci** kontroluje dwie różne właściwości zmiennej:
>   - Okres istnienia - jak długo zmienna może istnieć
>   - Zasięg - która część programu ma do niej dostęp
>
> Wyróżniamy 4 rodzaje klas:

Zmienna lokalna

> **Zmienna lokalna / automatyczna** - istnieje tylko wewnątrz danej funkcji, życie zmiennej kończy się, gdy funkcja kończy działanie

```cpp
int main()
{
    int var = 5; // Zmienna lokalna
    cout << var;

    return 0;
}
```

Zmienna globalna

Zmienna globalna - zdefiniowana poza wszystkimi funkcjami, jej zakras obejmuje cały program, życie zmiennej kończy się, gdy program kończy działanie

```cpp
int var = 5; // Zmienna globalna

void test();

int main()
{
    cout << var << endl;
    test();

    return 0;
}

void test()
{
    cout << var;
}
```

Statyczna zmienna lokalna

> **Statyczna zmienna lokalna** - istnieje tylko wewnątrz danej funkcji, ale jej życie kończy się, gdy program kończy działanie

```cpp
void test()
{
```

```
    static int var = 0; // Statyczna zmienna lokalna, podczas drugiego
wywołania zmienna var już istnieje z wartością 1 i to do niej zostanie dodana 1
    ++var;

    cout << var << endl;
}

int main()
{
    test();
    test();
    return 0;
}

Wynik:
1
2
```

Pamięć lokalna wątku

> **Pamięć lokalna wątku** - magazyn lokalny wątku to mechanizm, za pomocą którego zmienne są przydzielane w taki sposób, że na każdy istniejący wątek przypada jedno wystąpienie zmiennej. W tym celu używane jest słowo kluczowe `thread_local` - dużo mi to mówi

Rekurencja

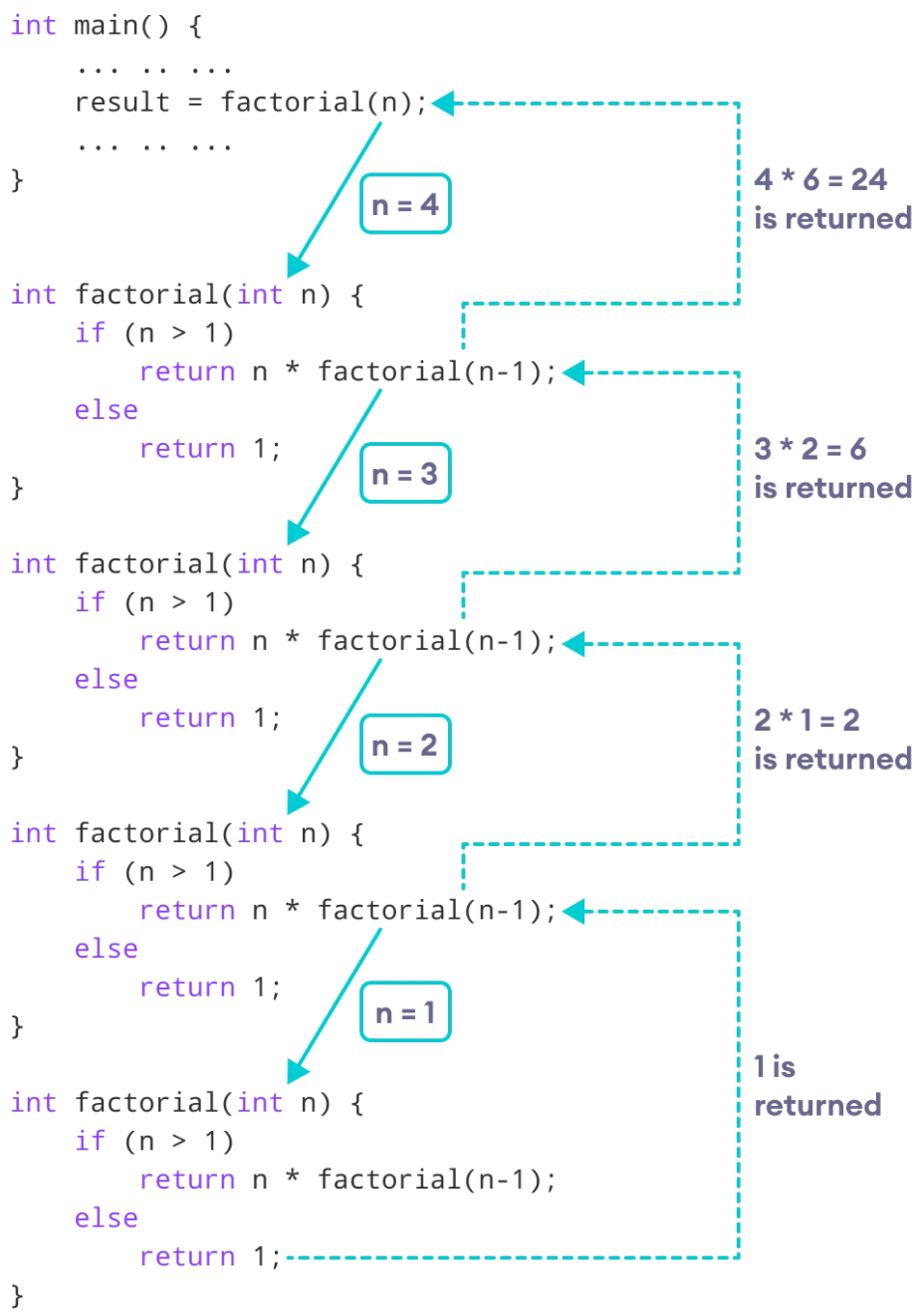
> Funkcja rekurencyjna - funkcja która wywołuje samą siebie - tworzy swoje klony do obliczenia wyniku - wskutek czego jest bardziej zasobożerny niż iteracja
>
> - Trzeba zadbać by za każdym wywołaniem funkcji problem się redukuje
> - Przydatna wiedza z ciągów - matematyka
>   **Zalety**: kod jest krótki i przejrzysty, wymagana w przypadku problemów dotyczących struktur danych i zaawansowanych algorytmów
>   **Wady:** duże zużycie zasobów komputera, trudniejsze debugowanie

```
void rekurencja() // Deklaracja funkcji
{
    ... .. ...
    rekurencja(); // Funkcja wywołuje samą siebie do obliczenia
    ... .. ...
}

int main()
{
    ... .. ...
    rekurencja(); // Wywołanie funkcji
    ... .. ...
}
```

Przykład - Silnia liczby przy użyciu rekursji:

```
int oblicz_silnie(int);

int main()
{
    int n, silnia;

    cout << "Wpisz liczbe nieujemna: ";
    cin >> n;

    silnia = oblicz_silnie(n); // Przypisanie wartości zwracanej przez funkcję
    cout << "Silnia z " << n << " = " << silnia;
    return 0;
}

int oblicz_silnie(int n) // Deklaracja funkcji
{
    if (n > 1) // Warunek podstawowy - silnia - !0 = 1
        return n * oblicz_silnie(n - 1); // Jeśli n jest większe od 1, funkcja wywołuje samą siebie tyle że z n - 1, proces się powtarza aż do chwli, gdy n = 1 i funkcja zwraca 1 - co powoduje wykonanie się wcześniejszych klonów funkcji i zwrócenie wyniku
    else
        return 1;  // Warunek podstawowy - silnia - !0 = 1
```

```
}

/* lub zamienione miejscami */

int oblicz_silnie(int n) // Deklaracja funkcji
{
    if (n < 1) // Warunek podstawowy - silnia - !0 = 1
        return 1;
    else
        return n * oblicz_silnie(n - 1); // W przeciwnym wypadku silnia =
licznba * Wywołanie funkcji przez samą siebie i redukcja -1 wartość liczby -
proces się powtarza, n = 0
}
```



PRZYKŁAD: Rozwiązanie iteracyjne:

```
int oblicz_silnie(int);

int main()
{
    int n, silnia;

    cout << "Wpisz liczbe nieujemna: ";
```

```
    cin >> n;

    silnia = oblicz_silnie(n); // Przypisanie wartości zwracanej przez funkcje
    cout << "Silnia z " << n << " = " << silnia;
    return 0;
}

int oblicz_silnie(int n) // Deklaracja funkcji
{
    int silnia = 1;
    for (int i = 1; i <= n; i++) // Iteracje
    {
        silnia *= i;
    }
    return silnia;
}
```

Powrót przez odniesienie

> Powrót przez odniesienie

```
int num; // Globalna zmienna

int &test(); // Prototyp funkcji

int main()
{
    test() = 5; // Zwraca adres num więc można przypisać jej wartość

    cout << num;
    test();
    return 0;
}

int &test() // Deklaracja funkcji
{
    return num; // Zwraca adres zmiennej globalnej i przypisuje 5 zadeklarowane
```

```
w funkcji main, tak więc num = 5
}
```

Tablice i łańcuchy w C++

Tablice

> **Tablica** - zmienna, która może rzechowywać wiele wartości tego samego typu

- Ilość elementów tablicy musi mieć wartość stałą - brak możliwości zmiany po deklaracji tablicy
- Uwaga na rozmiar tablicy
  - Numeracja indeksów zaczyna sie od 0 - np.: `tab[5]` → |0|1|2|3|4|
  - Nieprzekraczać wartości tablicy - inaczej błąd krytyczny
  - Jeśli `tab[n]` to ostatni element jest przechowywany pod indeksem `(n-1)`
- Elementy tablicy mają **kolejne adresy**. Na przykład załóżmy, że adres początkowy `x[0]` to `2120d`. Wówczas adres następnego elementu `x[1]` będzie miał wartość `2124d`, adres `x[2]` to `2128d` itd.

```
typ_danych nazwa_tabliczy[rozmiartablicy];
PRZTKŁAD:
int tab[6]; // Deklaracja tablicy

/* lub */

int x = 6;
int tab[x]; // Deklaracja tablicy ze zmienną np.: lokalną
```

> **Inicjalizacja tablicy** - wypełnienie tablicy przy deklaracjii

```
int tab[6] = {1, 3, 5, 7, 9, 11}; // Inicjalizacja tablicy

/* lub */

int x[] = {1, 3, 5, 7, 9, 11}; // Przypisanie automatyczne wielkości tablicy,
```

```
wynikającej z liczby przypisanych elementów

/* lub */

int tab[6] = {1, 3, 5}; // Inicjalizacja tablicy z pustymi członkami
```

> **Dostęp do elementów w tablicy** - każdy element tablicy jest powiązany z liczbą. Liczba jest nazywana **indeksem tablicy**. Możemy uzyskać dostęp do elementów tablicy za pomocą tych indeksów

```
// Indeksy = |0 |1 |2 |3 |4 |5 |
int tab[6] = {1, 3, 5, 7, 9, 11}; // Inicjalizacja

cout << tab[0]; // Wywołanie danej komórki tablicy
cout << tab[5];

Wynik:
1
11
```

> Pobieranie i wyprowadzanie danych

```
PRZYKŁAD_1 - zwykłe pętle:
    int tab[5];

    for (int i = 0; i < 5; i++) // Zwykła pętla pobierająca elementy tablicy
    {
        cout << "Podaj liczbe: ";
        cin >> tab[i];
    }

    for (int i = 0; i < 5; ++i) // Zwykła pętla wypisująca elementy tablicy
    {
        cout << tab[i] << " ";
    }
```

```
PRZYKŁAD_2 - pętle odwołujące się do adresu elemetu tabeli - oszczędza zasoby

    int tab[5];

    for (int i = 0; i < 5; i++) // Zwykła pętla pobierająca elementy tablicy
    {
        cout << "Podaj liczbe: ";
        cin >> tab[i];
    }

    for (int &n : tab) // Pętla pobierająca adres elementów i wypisująca wartości z możliwością ich zmiany
    {
        cout << n << " ";
        n = n + 1; // Zmiana zawartości elementów tablicy
    }

    cout << endl;

    for (const int &n : tab) // Pętla pobierająca adres elementów i wypisująca wartości bez możliwości ich zmiany - const
    {
        cout << n << " ";
        // n = n + 1; -> Error
    }
```

Tablice wielowymiarowe

> Całkowita liczba elementów tabilcy wielowymiarowej obliczomu mnożąc jej wymiary np.: `3 x 4 = 12` , `2 x 4 x 3 =24`

```
int tab[3][4]; // Deklaracja tablicy dwuwymiarowej
float tab[2][4][3]; // Deklaracja tablicy trzywymiarowej itd.
```

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Inicjalizacja tablicy wielowymiarowej

```
int tab[2][3] = {{1, 2, 3},
                 {4, 5, 6}};

int tab[3][2] = {{1, 2},
                 {3, 4},
                 {5, 6}};

int tab[2][3][4] = {{{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
                    {{13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9}}};
```

Pobieranie i wyprowadzanie danych

```
int tab[2][3][2];

for (int i = 0; i < 2; i++) // Pętla pobierająca
{
    for (int j = 0; j < 3; j++)
    {
        for (int k = 0; k < 2; k++)
        {
            cin >> tab[i][j][k];
```

```
            }
        }
    }

    for (int i = 0; i < 2; i++) // Pętla wypisująca
    {
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 2; k++)
            {
                cout << "tab[" << i << "][" << j << "][" << k << "] = " <<
tab[i][j][k] << endl;
            }
        }
    }
```

Funkcje i tablice

```
typ_zmiennej nazwa_funkcji(typ_argumentu nazwa_tablicy[rozmiar tablicy]) {
    // code
}

PRZYKŁAD:
int funkcja(int tab[5]) {
    // code
}
```

```
PRZYKŁAD:
// Określenie liczby wierszy w tablicy nie jest obowiązkowe. Jednak zawsze
należy określić liczbę kolumn - dlatego int n[][2]
void display(int n[][2]) // Definicja funkcji + przekazanie tablicy 2d
{
    cout << "Wyswietl wartosc: " << endl;
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 2; ++j)
        {
            cout << "num[" << i << "][" << j << "]: " << n[i][j] << endl;
        }
```

```
    }
}

int main()
{
    int num[3][2] = {
        {3, 4},
        {9, 5},
        {7, 1}};

    display(num);

    return 0;
}
PS.: Wskaźniki do tablic później
```

Łańcuchy String i Char

> Łańcych - zbiór znaków
> `Stringi` - tablice typu `char` zakończone znakiem `null`, to znaczy `\0` (wartość znaku null w kodzie ASCII wynosi 0)

```
PRZYKŁADY:
char str [] = "C ++";
char str [4] = "C ++";
char str [] = {'C', '+', '+', '\ 0'};
char str [4] = {'C', '+', '+', '\ 0'};
char str [100] = "C ++";
string str;
string str = "C ++";

PRZYKŁAD_1 - łą:
    char str[100];

    cout << "Wprowadz lancych: ";
    cin >> str; // Łańcych kończy się po wprowadzeniu spacji
    cout << "Wprowadziles: " << str << endl;
```

```
    cout << "\nWprowadz lancych: ";
    cin >> str;
    cout << "Wprowadziles: " << str << endl;

PRZYKŁAD_2:
    char str[100];
    cout << "Wprowadz lancych: ";
    cin.get(str, 100); // Łańcych pobiera całą linie

    cout << "Wprowadziles: " << str << endl;

PRZYKŁAD_3:
    string str;

    cout << "Wprowadz lancych: ";
    cin >> str; // Łańcych kończy się po wprowadzeniu spacji
    cout << "Wprowadziles: " << str << endl;

    cout << "\nWprowadz lancych: ";
    cin >> str;
    cout << "Wprowadziles: " << str << endl;

PRZYKŁAD_4:
    string str;
    cout << "Wprowadz lancych: ";
    getline(cin, str); // Łańcych pobiera całą linie

    cout << "Wprowadziles: " << str << endl;
PS.: Przykład 1 i 3 - rozwiązanie problemu spacji w Zabezpieczenie przed
wprowadzaniem błędnych danych
```

> Funkcje i łańcychy

```
void display(char *); // * nietrzeba tej gwiazdki
void display(string);

int main()
{
```

```
    string str1;
    char str[100];

    cout << "Wprowadz lancuch string: ";
    getline(cin, str1);

    cout << "Wprowadz lancuch char: ";
    cin.get(str, 100, '\n');

    display(str1);
    display(str);
    return 0;
}

void display(char s[])
{
    cout << "Wprowadzony lancuch char: " << s << endl;
}

void display(string s)
{
    cout << "Wprowadzony lancuch string: " << s << endl;
}
```

https://www.programiz.com/cpp-programming/structure
https://www.youtube.com/watch?v=zSCOSjnFe4I&list=PL_1QM_dtCJDcV4qcY4Y2k8vk0thByI2vJ&index=15&ab_channel=Aitra