

# Podstawy Programowania Komputerów

Wykład 9

## Złożone typy danych

Rok akademicki: 2022/2023



Politechnika  
Śląska



UCZELNIA  
BADAWCZA  
INICJATYWA DOSKONAŁOŚCI

Dr inż. Łukasz Maliński  
[lukasz.malinski@polsl.pl](mailto:lukasz.malinski@polsl.pl)



- Typ wyliczeniowy
- Aliasy typów
- Struktury
- Operacje wskaźnikowe ze strukturami
- Praktyczne wskazówki jak dobierać pola w strukturze
- Struktura jako wartość zwracana
- Tablica struktur
- Definiowanie znaczenia operatorów dla typów własnych

**Typ wyliczeniowy** służy do przechowywania informacji, które przyjmują wartości ze **skończonego i ściśle zdefiniowanego zbioru**.

## Przykłady:

Płeć  $\in \{\text{kobieta, mężczyzna}\},$

Włącznik  $\in \{\text{włączony, wyłączony}\},$

Działanie  $\in \{\text{dodawanie, odejmowanie, mnożenie, dzielenie}\},$

Kot  $\in \{\text{żywy, martwy, piekielnie\_wściekły}\}^*.$

## Składnia definicji:

```
enum nazwa_typu
```

```
{  
    OPCJONALNE → wartosc_1 = 0,  
    wartosc_2 = 1,  
    wartosc_3 = 5,  
}; ← ŚREDNIK!  
      ↑  
      PRZECINKI!
```

## Deklaracja instancji:

```
nazwa_typu nazwa_instancji;
```

## Użycie:

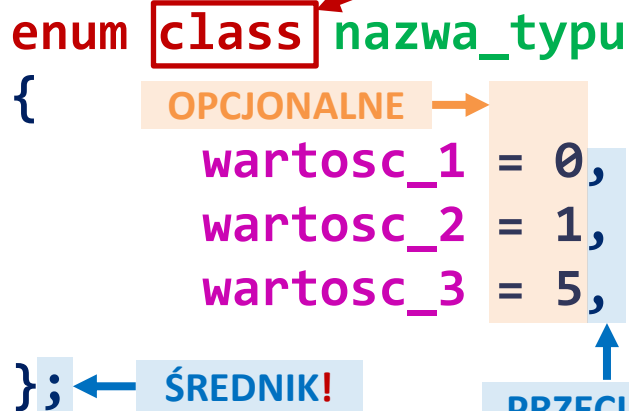
```
nazwa_instancji = wartosc_1;
```

## UWAGI:

- Definicję zaczynamy od **enum**.
- Po klamrze zamykającej listę wartości **musimy** dać średnik.
- Kolejne wartości rozdzielamy przecinkami.
- Przypisanie numeracji do wartości, jest opcjonalne.
- Instancję typu wyliczeniowego, tworzymy w analogiczny sposób jak zwykłą zmienną typu fundamentalnego.
- Przypisanie wartości odbywa się po jej nazwie.
- **Wada:** nazwy wartości mają zasięg globalny, więc mogą wejść w konflikt z innymi nazwami w programie.

## Składnia definicji:

```
enum class nazwa_typu  
{  
    OPCJONALNE wartosc_1 = 0,  
    wartosc_2 = 1,  
    wartosc_3 = 5,  
};
```



NOWE  
ELEMENTY  
SKŁADNI

## UWAGI:

- Przy definicji dopisujemy słowo **class**.
- Reszta definicji, oraz sposób tworzenia instancji, są takie same jak przy **enum** z C.
- Przypisując wartość **musimy** podać nową typu i użyć operatora zakresu „**::**”, przed **nazwą wartości**.
- **ZALETY:** wartości mają zakres ważności ograniczony tylko do tego konkretnego typu, więc nie grozi nam konflikt nazw.
- **WADA:** Musimy za każdym razem podać, o który typ wyliczeniowy nam chodzi.

## Deklaracja instancji:

```
nazwa_typu nazwa_instancji;
```

## Użycie:

```
nazwa_instancji = nazwa_typu::wartosc_1;
```



# Typ wyliczeniowy z kluczem liczbowym

6

```
1) enum class DayOfWeek
2) {
3)     First      = 1,
4)     Monday     = 1,
5)     Tuesday    = 2,
6)     Wednesday  = 3,
7)     Thursday   = 4,
8)     Friday     = 5,
9)     Saturday   = 6,
10)    Sunday     = 7,
11)    Last        = 7,
12) };

13) void main(void)
14) {
15)     DayOfWeek dzien;
16)     int numer;
17)     cout << "Podaj dzien tygodnia [1-7]: ";
18)     cin >> numer;

19)     dzien = (DayOfWeek)numer;
20)     if (dzien == DayOfWeek::First)
21)         cout << "Pierwszy dzien tygodnia" << endl;

22)     int pozostalo = (int)DayOfWeek::Last - (int)dzien;
23)     cout << "Pozostalo " << pozostalo
24)         << " dni do konca tygodnia" << endl;
25) }
```

**PRZYPISANIE KLUCZA LICZBOWEGO**

**ZDUBLOWANIE KLUCZA**

**PRZYPISANIE PRZEZ RZUTOWANIE**

**OBLICZENIA PRZEZ RZUTOWANIE**

## WAŻNE:

- + przypisanie klucza liczbowego pozwala na łatwe rzutowanie do typu *int* (klucz nie musi być nawet kolejnymi liczbami),
- + można przypisać ten sam klucz do różnych nazw, tworząc aliasy wartości,
- + dzięki przypisaniu klucza liczbowego przypisywanie wartości można zrealizować na rzutowaniu
- + dzięki rzutowaniu można na typie wyliczeniowym wykonywać też obliczenia jak na liczbach.

C:\Windows\system32\cmd.exe

```
Podaj dzien tygodnia [1-7]: 1
Pierwszy dzien tygodnia
Pozostalo 6 dni do konca tygodnia
```

# Typ wyliczeniowy z kluczem znakowym

7

```
1) enum class Dzialanie
2) {
3)     dodawanie = '+',          PRZYPISANIE
4)     odejmowanie = '-',        KLUCZA
5)     mnozenie = '*',           ZNAKOWEGO
6)     dzielenie = '/',
7) };

8) void main(void)
9) {
10)     Dzialanie operacja;
11)     char znak; double a, b, c;
12)     cout << "Wybierz dzialanie [+,-,*,/]: ";
13)     cin >> znak;

14)     operacja = (Dzialanie)znak;    RZUTOWANIE

15)     cout << "Podaj a: "; cin >> a;
16)     cout << "Podaj b: "; cin >> b;

17)     switch (operacja)              PRZETWARZANIE
18)     {
19)     default:
20)     case Dzialanie::dodawanie:
21)         c = a + b; break;
22)     case Dzialanie::odejmowanie:
23)         c = a - b; break;
24)     case Dzialanie::mnozenie:
25)         c = a * b; break;
26)     case Dzialanie::dzielenie:
27)         c = a / b; break;
28)     }
29)     cout << "Wynik: " << c << endl;

30) }
```

## WAŻNE:

- + do wartości, można także przypisać klucz znakowy (typ *char* to też liczba całkowita),
- + przepisywanie znaków do wartości typu wyliczeniowego przez rzutowanie jest równie proste jak w przypadku liczb,
- + operacje na typie wyliczeniowym w obrębie switch-case są znacznie czytelniejsze (samokomentujące),
- + typ wyliczeniowy tak naprawdę przerzuca problem kodowania/dekodowania różnych wartości na kompilator, uwalniając nas o tworzenia dokumentacji z tabelkami kodowania tych wartości.

C:\Windows\system32\cmd.exe

```
Wybierz dzialanie [+,-,*,/]: +
Podaj a: 3
Podaj b: 7
Wynik: 10
```


- 1) Poprzez makrodefinicję:** STANOWCZO ODRADZANY

(preprocesor podmienia **nazwę aliasu** **nazwą typu** przed kompilacją)

Składnia: **#define** **Nazwa\_aliasu** **Nazwa\_typu**

Przykład: **#define** **Plec** **bool**

BRAK ŚREDNIKA!


- 2) Klasyczny sposób z języka C:** DOPUSZCZALNY

(definiuje nowy alias typu danych)

Składnia: **typedef** **Nazwa\_typu** **Nazwa\_aliasu**;

Przykład: **typedef** **double** **Odlglosc**;
- 3) Nowy sposób z C++11:** ZALECANY

(definiuje nowy alias typu danych, kompatybilny z szablonami – sem. 3)

Składnia: **using** **Nazwa\_aliasu** = **Nazwa\_typu**;

Przykład: **using** **Obrazenia** = **int**;



# Aliasy typów do zwiększania czytelności

9

```
1) int rzut(double d, int mod)
2) {
3)     int ob = 100 + mod;
4)     if (d > 100.0)
5)         return 0;
6)
7)     else if (d > 60.0)
8)         return (int)(0.4 * ob);
9)
10)    else if (d > 30.0)
11)        return (int)(0.6 * ob);
12)
13)    else
14)        return ob;
15) }
```

**NIECZYTELNE  
I NIEZROZUMIAŁE**

## WAŻNE:

+ czytelność i zrozumiałość obu fragmentów kodu,  
+ możliwość zmiany typu tylko w jednym miejscu w kodzie.

```
1) using Dystans = double;
2) using Obrazenia = int;
3) using Mod = double;
4) constexpr Dystans SHORT_DISTANCE = 10.0;
5) constexpr Dystans MEDIUM_DISTANCE = 30.0;
6) constexpr Dystans LONG_DISTANCE = 60.0;
7) constexpr Dystans MAXIMUM_DISTANCE = 100.0;
8) constexpr Obrazenia STD_DMG = 100;
9) constexpr Mod MEDIUM_DIS_MOD = 0.8;
10) constexpr Mod LONG_DIS_MOD = 0.6;
11) constexpr Mod MAXIMUM_DIS_MOD = 0.4;
12) Obrazenia rzut(Dystans odleglosc, Obrazenia modyfikator)
13) {
14)     Obrazenia obrazenia = STD_DMG + modyfikator;
15)     if (odleglosc > MAXIMUM_DISTANCE)
16)         return 0;
17)
18)     else if (odleglosc > LONG_DISTANCE)
19)         return (Obrazenia)(MAXIMUM_DIS_MOD * obrazenia);
20)
21)     else if (odleglosc > MEDIUM_DISTANCE)
22)         return (Obrazenia)(LONG_DIS_MOD * obrazenia);
23)
24)     else // (odleglosc <= SHORT_DISTANCE)
25)         return obrazenia;
26) }
```

**DEFINICJE ALIASÓW**

**DEFINICJE  
STAŁYCH**

**CZYTELNE  
I ZROZUMIAŁE**

# Aliasy typów do upraszczania definicji

10

```
1) enum class Bierka
2) {
3)     pusta    = 0, pionek  = 1, skoczek = 2, goniec  = 3,
4)     wieza    = 4, hetman  = 5, krol    = 6,
5) };
6) const char* const NAZWY_BIEREK[]
7)     = { "Pusta", "Pionek", "Skoczek",
8)         "Goniec", "Wieza", "Hetman", "Krol" };
9) using Szachownica = Bierka[8][8]; DEFINICJA ALIASU
10) Bierka typ_bierki(Szachownica&s, int x, int y)
11) {
12)     return s[x][y]; ODCZYT BIERKI
13) }
14) void ust_bierke(Szachownica&s, int x, int y, Bierka b)
15) {
16)     s[x][y] = b; ZAPIS BIERKI
17) }
18) void main(void)
19) {
20)     const int LICZBA_SZACHOWNIC = 10;
21)     Bierka bierka;
22)     Szachownica plansze[LICZBA_SZACHOWNIC];
23)     std::memset(plansze, 0, LICZBA_SZACHOWNIC * sizeof(Szachownica));
24)     ust_bierke(plansze[0], 0, 0, Bierka::hetman);
25)     ust_bierke(plansze[6], 3, 5, Bierka::skoczek);
26)     cout << "Bierka na polu (0,0) planszy [0]: "
27)          << NAZWY_BIEREK[(int)typ_bierki(plansze[0], 0, 0)] << "\n\n";
28)     cout << "Bierka na polu (3,5) planszy [6]: "
29)          << NAZWY_BIEREK[(int)typ_bierki(plansze[6], 3, 5)] << "\n\n";
30) }
```

**UŻYCIĘ TABLICZ SZACHOWNIC**

## WAŻNE:

- + definicja aliasu do zwykłej tablicy dwuwymiarowej,
- + posługiwanie się wszędzie aliasem,
- + tworzymy tu tak naprawdę tablicę 3D, ale dwa wymiary są jakby sztywno wpisane w „typ” Szachownica, a trzeci jest zadawany jako jej rozmiar.
- + łączenie typu wyliczeniowego z aliasem,
- + stosowanie aliasu nie tylko zwiększa czytelność kodu, ale także upraszcza znacząco jego składnię.

C:\Windows\system32\cmd.exe

```
Ile gier szachowych chcesz obserwować: 2
Bierka na polu (0,0) planszy [0]: Hetman
```

## Docelowy typ danych:

Tablica 10 wskaźników do funkcji przyjmujących i zwracających nieruchomy wskaźnik do tekstu.

## Bezpośrednia definicja bytu:

```
char * const (*tablica[10])(char * const);
```

NAZWA INSTANCJI



## Definicja z użyciem bezpośredniego aliasu:

```
using NowyTyp = char * const (*[10])(char * const);  
NowyTyp tablica;
```

BRAK NAZWY!



## Definicja z użyciem pośrednich aliasów:

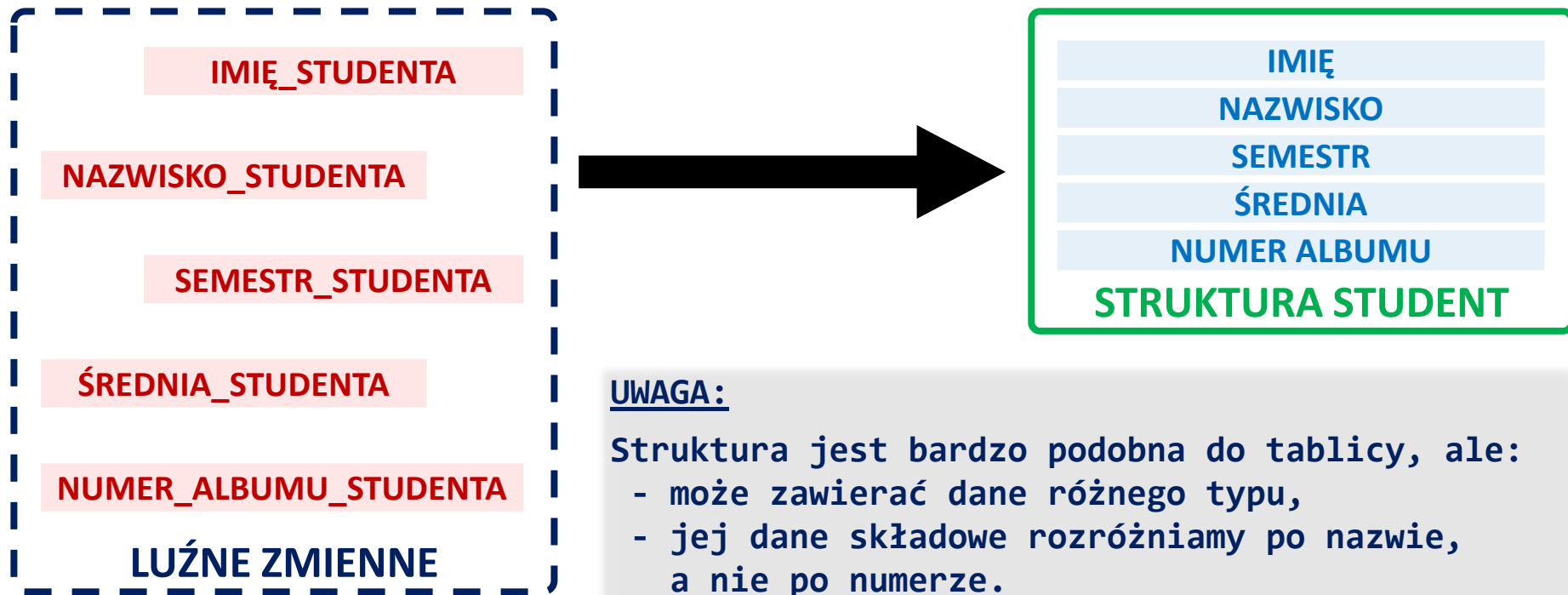
```
using WskNaFun = char * const (*) (char *const);  
using Nowy_typ = WskNaFun[10];  
Nowy_typ tablica;
```

ALIAS POŚREDNI



**Struktura** to złożony typ danych grupujący logicznie ściśle powiązane ze sobą informacje.

Przykład:



UWAGA:

Struktura jest bardzo podobna do tablicy, ale:

- może zawierać dane różnego typu,
- jej dane składowe rozróżniamy po nazwie, a nie po numerze.

## Definicja struktury:

**struct** NazwaTypu

```
{  
    typ_danych1 nazwaPola1;  
    typ_danych2 nazwaPola2;  
    ...  
    typ_danychN nazwaPolaN;  
};
```



ŚREDNIKI!

## Deklaracja instancji:

NazwaTypu nazwaInstancji;

## Dostęp do pola w instancji:

Zapis : nazwaInstancji.nazwaPola1 = wartość;

Odczyt: wartość = nazwaInstancji.nazwaPola1;

## UWAGI:

- Definiując strukturę definiujemy nowy złożony typ danych.
- Zmienne składowe struktury nazywamy **polami**.
- Każde pole **musi** mieć unikatową nazwę w obrębie struktury.
- Każde pole może być innego typu i może też być to typ złożony.
- Typy pól mogą się powtarzać.
- Po klamrze zamykającej ciało struktury **musimy** dać średnik.
- Instancje tworzymy tak jak inne zmienne.

```
1) using text = char[50];
2) const char* const PLEC[] = { "kobieta", "meczczyna" };
3) enum class Plec { kobieta, mezczyzna };
                                     DEFINICJE POMOCNICZE
4) struct Pacjent
5) {
6)     text imie, nazwisko;
7)     int  wiek;
8)     Plec plec;
9) };
                                     DEFINICJA STRUKTURY
                                     DO OPISU PACJENTA
10) void main(void)
11) {
12)     Pacjent pac01;
13)     strcpy(pac01.imie, "Marianna");
14)     strcpy(pac01.nazwisko, "Kowalska");
15)     pac01.wiek = 80;
16)     pac01.plec = Plec::kobieta;
                                     Utworzenie
                                     instancji
                                     i wypełnienie jej
                                     ręcznie: pole po polu
17)     Pacjent pac02 = { "Jan", "Nowak", 45, Plec::meczczyna };
                                     Utworzenie instancji z inicjalizacją
18)     cout << "Pacjent:\t" << pac01.imie << " " << pac01.nazwisko
19)         << endl << "\t\t" << PLEC[(int)pac01.plec]
20)         << " (" << pac01.wiek << " lat)." << endl << endl;
21)     cout << "Pacjent:\t" << pac02.imie << " " << pac02.nazwisko
22)         << endl << "\t\t" << PLEC[(int)pac02.plec]
23)         << " (" << pac02.wiek << " lat)." << endl << endl;
24) }
```

WYPISYWANIE ZAWARTOŚCI INSTANCJI

## WAŻNE:

- + definiując pola struktury można użyć dowolnych, fundamentalnych typów danych, aliasów, a nawet innych typów złożonych (*enum*, *struct*),
- + instancję można zainicjalizować przez podanie w {} wartości kolejnych jej pól, ale można to zrobić w ten sposób tylko przy jej tworzeniu,
- + możemy utworzyć wiele instancji tej samej struktury i każda z nich będzie mogła przechowywać inne wartości swoich pól,
- + dostęp do pól struktury uzyskujemy operatorem kropki „.”.

C:\Windows\system32\cmd.exe

```
Pacjent:      Marianna Kowalska
               kobieta (80 lat).

Pacjent:      Jan Nowak
               mezczyzna (45 lat).
```

```
1) using text = char[50];
2) enum class Plec { kobieta, mezczyzna };

3) struct PacjentA
4) {
5)     text imie, nazwisko;
6)     int  wiek;
7)     Plec plec;
8) };
    DEFINICJA STRUKTURY
    BEZ WARTOŚCI DOMYŚLNYCH

9) struct PacjentB
10) {
11)     text imie = "Jan" , nazwisko = "Nowak";
12)     int  wiek = 30;
13)     Plec plec = Plec::mezczyzna;
14) };
    DEFINICJA STRUKTURY
    Z WARTOŚCIAMI DOMYŚLNYMI

15) void main(void)
16) {
17)     PacjentA osoba1 = { "Teodor", "Kowalski", 25,
18)                        Plec::mezczyzna };
    OK

19)     PacjentB osoba2 = { "Marianna", "Lipa", 83,
20)                        Plec::kobieta };
    BŁĄD

21)     PacjentB osoba3;
22)     strcpy (osoba3.imie, "Janina");
23)     strcpy (osoba3.nazwisko, "Potocka");
24)     osoba3.wiek = 10;
25)     osoba3.plec = Plec::kobieta;
    OK
}
```

## WAŻNE:

- + pola w strukturze definiować można z, lub bez podania dla nich wartości domyślnych,
- + podanie wartości domyślnych sprawia, że każda nowa instancja będzie miała po utworzeniu pola o tych wartościach,
- + instancję struktury bez zdefiniowanych wartości domyślnych pól, można bez problemu inicjalizować przy deklaracji przy pomocy {},
- + kolejność i typy wartości muszą odpowiadać kolejności i typom pól w strukturze),
- + instancji struktury ze zdefiniowanymi wartościami domyślnymi, nie można tak łatwo poddać inicjalizacji przy pomocy {}, co implikuje tu ręczne wypełnianie ich treścią.

// DEKLARACJE POMOCNICZE JAK W POPRZEDNIM PRZYKŁADZIE

```
1) struct Pacjent
2) {
3)     text imie, nazwisko;
4)     int  wiek;
5)     Plec plec;
6) };
```

DEFINICJA STRUKTURY  
DO OPISU PACJENTA

```
7) void podajDane(Pacjent &p, char* im, char* naz, int w, Plec pl)
8) {
9)     strcpy(p.imie, im);
10)    strcpy(p.nazwisko, naz);
11)    p.wiek = w;
12)    p.plec = pl;
13) }
```

FUNKCJA  
DO ZAPISU INFORMACJI W INSTANCJI

```
14) void wypiszDane(Pacjent &p)
15) {
16)     cout << "Pacjent:\t" << p.imie << " " << p.nazwisko
17)     << endl << "\t\t" << PLEC[(int)p.plec]
18)     << " (" << p.wiek << " lat)." << endl << endl;
19) }
```

FUNKCJA DO WYPISU ZAWARTOŚCI INSTANCJI

```
20) void main(void)
21) {
22)     Pacjent pac01, pac02;
23)
24)     podajDane(pac01, "Marianna", "Kowalska", 80, Plec::kobieta);
25)     podajDane(pac02, "Jan", "Nowak", 45, Plec::meczczynna);
26)
27)     wypiszDane(pac01);
28)     wypiszDane(pac02);
29) }
```

UŻYCIE FUNKCJI

## WAŻNE:

- + **kluczowe** jest przekazanie instancji do procedury przez referencję, lub przez wskaźnik, aby pracować na orginalie,
- + przekazanie **do funkcji zapisującej** przez wartość byłoby poważnym błędem, gdyż wpisana treść trafiłaby tylko do lokalnej kopii,
- + przekazanie **do funkcji wyświetlającej** przez wartość, nie byłoby poważnym błędem, ale odbyło by się kosztem wydajności kodu.
- + **proceduralne przetwarzanie instancji struktur** można wykonać w dowolnej chwili (jeśli tylko instancja istnieje).

C:\Windows\system32\cmd.exe

```
Pacjent:      Marianna Kowalska
               kobieta (80 lat).
```

```
Pacjent:      Jan Nowak
               mezczyzna (45 lat).
```



// DEKLARACJE POMOCNICZE JAK W POPRZEDNIM PRZYKŁADZIE

```
1) struct Pacjent
2) {
3)     text imie, nazwisko;
4)     int  wiek;
5)     Plec plec;
6) };

7) void podajDane(Pacjent *p, char* im, char* naz, int w, Plec pl)
8) {
9)     strcpy(p->imie, im); strcpy(p->nazwisko, naz);
10)    p->wiek = w; p->plec = pl;
11) }

12) void wypiszDane(Pacjent *p)
13) {
14)     cout << "Pacjent:\t" << p->imie << " " << p->nazwisko
15)         << endl << "\t\t" << PLEC[(int)p->plec]
16)         << " (" << p->wiek << " lat)." << endl << endl;
17) }

18) void main(void)
19) {
20)     Pacjent pac01, pac02;

21)     podajDane(&pac01, "Marianna", "Kowalska", 80, Plec::kobieta);
22)     podajDane(&pac02, "Jan", "Nowak", 45, Plec::meczyczna);

23)     wypiszDane(&pac01); wypiszDane(&pac02);
24) }
```

\* ZAMIAST &

-> ZAMIAST .

& PRZED  
NAZWĄ INSTANCJI

## WAŻNE:

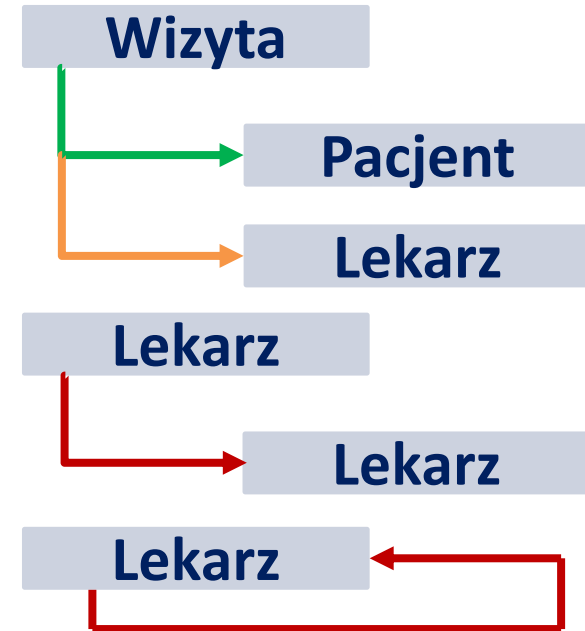
- + przesyłanie przez wskaźnik jest równie skuteczne co przez referencję, ale odrobinę mniej bezpieczne, gdyż wskaźnik można w obrębie funkcji przestawić, a referencji nie,
- + do pól instancji wskazanej przez wskaźnik, dostajemy się przy pomocy operatora „->” a nie „.”,
- + „ wsk -> pole ” to zapis skrócony od „ (\*wsk).pole.”
- + wywołując funkcję musimy pamiętać o przekazaniu adresu instancji (& przed nazwą).

C:\Windows\system32\cmd.exe

```
Pacjent:      Marianna Kowalska
               kobieta (80 lat).

Pacjent:      Jan Nowak
               mezczyzna (45 lat).
```

```
1) using text = char[50];
2) struct Pacjent
3) {
4)     text imie, nazwisko;
5)     int wiek;
6)     Plec plec;
7) };
8) struct Lekarz
9) {
10)    text imie, nazwisko;
11)    int wiek;
12)    Plec plec;
13)    Lekarz const * ordynator;
14) };
15) struct Wizyta
16) {
17)    Pacjent const *pacjent;
18)    Lekarz const *lekarz;
19) };
```



## WAŻNE:

- + wskaźnik może być polem struktury,
- + wskaźnik może wskazywać instancję innego typu strukturalnego,
- + wskaźnik może także wskazywać ten sam typ strukturalny,

// DEFINICJE STRUKTUR JAK W POPRZEDNICH PRZYKŁADACH

```
1) void wypiszDane(Wizyta *w)
2) {
3)     cout << "Pacjent:\t" << w -> pacjent -> nazwisko << endl
        << "ma wizyte u: " << w -> lekarz -> nazwisko
        << endl << endl;
4) }

5) void main(void)
6) {
7)     Pacjent pac01 = { "Jan", "Nowak", 45, Plec::meczczynna};
8)     Pacjent pac02 = { "Marianna", "Kowalska", 80, Plec::kobieta};
9)     Lekarz ordynator = { "Antoni", "Wargacz", 60, Plec::meczczynna, &ordynator };
10)    Lekarz rezydent = { "Tomasz", "Soltys", 30, Plec::meczczynna, &ordynator };
11)    Wizyta wiz01 = { &pac01, &rezydent };
12)    Wizyta wiz02 = { &pac02, &ordynator };
13)    wypiszDane(&wiz01);
14)    wypiszDane(&wiz02);
15) }
```

**DWUPOZIOMOWY DOSTĘP DO PÓL**

**DEFINICJA ASCJACJI/AGREGACJI**

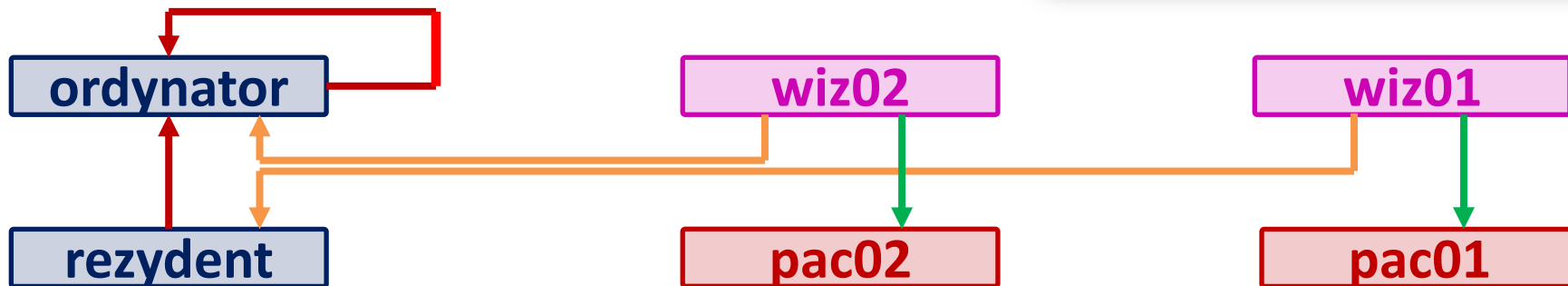
**WAŻNE:**

- + dwupoziomowy dostęp do pola,
- + zdefiniowanie asocjacji między instancjami przy pomocy wskaźników,
- + dzięki asocjacji/agregacji można mocno zaoszczędzić zużycie pamięci, gdyż dane nie są niepotrzebnie powielane.

C:\Windows\system32\cmd.exe

```
Pacjent:          Nowak
ma wizyte u: Soltys

Pacjent:          Kowalska
ma wizyte u: Wargacz
```



Jeśli opisywana informacja jest prosta i ściśle powiązana z daną instancją projektowanej struktury

np.: **wiek, nazwisko, płeć, stan aktywności zawodowej**,  
to zapisz ją jako zwykłe pole (zmienną/tablicę).

Przykład:

## Pracownik

char	nazwisko[30];	(tablica)
Plec	plec;	(enum)
int	wiek;	(zmienna)
bool	aktZawodowo	(stan)
ltp...		

```
1)    enum class Plec{kobieta,mezczyzna};  
2)    struct Pracownik  
3)    {  
4)        char nazwisko[30];  
5)        Plec plec;  
6)        int  wiek;  
7)        bool aktZawodowo;  
8)    };
```

Jeśli opisywana informacja jest złożona, ale ściśle powiązana z daną instancją projektowanej struktury

np. **w samochodzie: pojemność silnika i liczba cylindrów**,  
to zapisz ją jako instancję składową (utwórz osobną strukturę opisującą tą złożoną informację i jej instancję umieść w strukturze jako pole) – tzw. **Kompozycja**.

Przykład:

## Samochod

`int liczPasazerow`

(zwykłe pole)

### Silnik

`double pojemnosc`  
`int liczbaCylindrow`

(instancja  
składowa)

```
1) struct Silnik
2) {
3)     double pojemnosc;
4)     int liczbaCylindrow;
5) };
6) struct Samochod
7) {
8)     int liczPasazerow;
9)     Silnik silnik;
10) };
```

Jeśli opisywana informacja jest złożona, ale powinna istnieć niezależnie od instancji projektowanej struktury

np.: **polisa** (w strukturze Klient),

to zapisz ją jako pole wskaźnikowe, wskazujące na zewnętrzną instancję innej struktury - **asocjacja**.

Przykład:

## Klient

int nrID (zwykłe pole)  
Polisa \*umowa (pole wskaźnikowe)  
Agent \*prowadzacy (pole wskaźnikowe)

## Polisa

int nrEwidencyjny  
double wartosc

## PROTOTYP STRUKTURY

```
1) struct Agent;
2) struct Polisa
3) {
4)     int nrEwidencyjny;
5)     double wartosc;
6) };
7) struct Klient
8) {
9)     int nrID;
10)    Polisa *umowa;
11)    Agent *prowadzacy;
12) };
```

// DEFINICJE STRUKTUR I FUNKCJI JAK W POPRZEDNICH PRZYKŁADACH

```
1) void umowWizyte(Wizyta &w, Pacjent const &p, Lekarz const &l)
2) {
3)     w.pacjent = &p;
4)     w.lekarz = &l;
5) }
6) Wizyta umowWizyte(Pacjent const &p, Lekarz const &l)
7) {
8)     Wizyta result;
9)     result.pacjent = &p;
10)    result.lekarz = &l;
11)    return result;
12) }
13) void main(void)
14) {
15)     Pacjent pac01 = { "Jan", "Nowak", 45, Plec::meczczyna };
16)     Lekarz ordynator = { "Antoni", "Wargacz", 60, Plec::meczczyna, &ordynator };
17)     Lekarz rezydent = { "Tomasz", "Soltys", 30, Plec::meczczyna, &ordynator };
18)     Wizyta wiz01, wiz02;
19)     umowWizyte(wiz01, pac01, ordynator);
20)     wiz02 = umowWizyte(pac01, rezydent);
21)     wypiszDane(&wiz01); wypiszDane(&wiz02);
22) }
```

WYPEŁNIANIE WIZYTY PRZECZ  
MODYFIKACJĘ ARGUMENTU REFERENCYJNEGO

TWORZENIE NOWEJ WIZYTY  
I ZWRACANIE JEJ PRZECZ WARTOŚC

## WAŻNE:

- + pokazano dwa sposoby na wpisanie treści do instancji,
- + sposób ze zwracaniem instancji przez wartość (return) jest bardziej intuicyjny,
- + jak widać, dzięki strukturom można w funkcji zwrócić typ złożony, a zatem też efektywnie zwrócić więcej niż jedną wartość na raz przez return,
- + modyfikacja instancji przez referencję jest wydajniejsza na starych kompilatorach.

C:\Windows\system32\cmd.exe

```
Pacjent:          Nowak
ma wizyte u: Wargacz
```

```
Pacjent:          Nowak
ma wizyte u: Soltys
```

// DEFINICJE STRUKTUR I FUNKCJI JAK W POPRZEDNICH PRZYKŁADACH

```
1) void wypiszPacjentow(Wizyta const * wizyty, int n, Lekarz &lek)
2) {
3)     cout << "Wizyty dla " << lek.nazwisko << ":" << endl;
4)     int numWiz = 1;
5)     for (int i = 0; i < n; i++)
6)         if (wizyty[i].lekarz == &lek)
7)             cout << numWiz++ << ":" << wizyty[i].pacjent->nazwisko << endl;
8) }
```

FUNKCJA DO OPERACJI NA TABLICY STRUKTUR

```
9) void main(void)
10) {
11)     constexpr int LIMIT = 3;
12)     Pacjent pacjenci[] = { { "Jan", "Nowak", 45, Plec::mezczyzna },
13)                             { "Marianna", "Kowalska", 80, Plec::kobieta },
14)                             { "Marek", "Kuc", 20, Plec::mezczyzna } };
15)     Lekarz rezydenci[] = { {"Tomasz", "Soltys", 30, Plec::mezczyzna, nullptr},
16)                             {"Miłosz", "Rzeznik", 45, Plec::mezczyzna, nullptr} };
```

DEKLARACJA TABLICY STRUKTUR Z RECZNĄ INICJALIZACJĄ

```
14) Wizyta wizyty[LIMIT] = { umowWizyte(pacjenci[0], rezydenci[0]),
15)                             umowWizyte(pacjenci[1], rezydenci[1]),
16)                             umowWizyte(pacjenci[2], rezydenci[0]) };
```

WYWOŁANIA FUNKCJI NA LIŚCIE INICJALIZATORÓW TABLICY

```
15) wypiszPacjentow(wizyty, LIMIT, rezydenci[0]);
16) }
```

## WAŻNE:

- + deklaracja i inicjalizacja tablic struktur, jest bardzo podobna deklaracji i inicjalizacji tablic innych typów (trzeba tylko pamiętać o podwójnym {}),
- + podczas inicjalizacji możemy w {} także wywołać funkcję,
- + przetwarzanie tablic struktur pod względem składniowym nie różni się od przetwarzania tablic typów fundamentalnych,
- + odwołując się do pola struktury w tablicy, musimy najpierw wskazać element tablicy.

C:\Windows\system32\cmd.exe

```
Wizyty dla Soltys:
1: Nowak
2: Kuc
```



// DEFINICJE STRUKTUR I FUNKCJI JAK W POPRZEDNICH PRZYKŁADACH

```
1) Wizyta umowWizyte(Pacjent const &p, Lekarz const &l)
2) {
3)     Wizyta result;
4)     result.pacjent = &p;
5)     result.lekarz = &l;
6)     return result;
7) }

8) Wizyta operator+(Pacjent const &p, Lekarz const &l)
9) {
10)    Wizyta result;
11)    result.pacjent = &p;
12)    result.lekarz = &l;
13)    return result;
14) }

15) bool operator>(Pacjent const &left, Pacjent const &right)
16) {
17)     return left.wiek > right.wiek;
18) }

19) void main(void)
20) {
21)     Pacjent pacjenci[] = { { "Jan", "Nowak", 45, Plec::meczczyna },
22)                             { "Marianna", "Kowalska", 80, Plec::kobieta } };
23)     Lekarz rezydent = { "Tomasz", "Soltys", 30, Plec::meczczyna, nullptr };
24)
25)     Wizyta wiz01;
26)     if (pacjenci[0]>pacjenci[1])
27)         wiz01 = pacjenci[0] + rezydent;
28)     else
29)         wiz01 = pacjenci[1] + rezydent;
30)
31)     wypiszDane(&wiz01);
32) }
```

**ZAMIANA: umowWizyte NA operator+**

**DOKŁADNIE TO SAMO!**

**PORÓWNYWANIE PACJENTÓW (KRYTERIUM WIEKOWE)**

**UMAWIANIE WIZYT OPERATOREM +**

## WAŻNE:

- + w języku C++ można definiować jak powinny działać operatory dla zdefiniowanych przez nas typów danych,
- + to jak działają operatory zależy tylko od nas, ale powinniśmy zadbać o ich intuicyjność,
- + każdy operator oprócz () ma ściśle zdefiniowaną liczbę argumentów i musimy się do niej dostosować,
- + można definiować relację między instancjami typu: „==”, „!=”, „>”, „>=”, itd. (to też są operatory).

C:\Windows\system32\cmd.exe

```
Pacjent: Kowalska
ma wizyte u: Soltys
```

// DEFINICJE STRUKTUR I FUNKCJI JAK W POPRZEDNICH PRZYKŁADACH

```
1) void wypiszDane(Pacjent *p)
2) {
3)     cout << "Pacjent:\t" << p->imie << " " << p->nazwisko
         << endl << "\t\t" << PLEC[(int)p->plec]
         << " (" << p->wiek << " lat)." << endl << endl;
4) }

5) ostream& operator<<(ostream &str, Pacjent const &p)
6) {
7)     str << "Pacjent:\t" << p.imie << " " << p.nazwisko
         << endl << "\t\t" << PLEC[(int)p.plec]
         << " (" << p.wiek << " lat)." << endl << endl;
8)     return str;
9) }

10) void main(void)
11) {
12)     Pacjent pacjenci[] = { { "Jan", "Nowak", 45, Plec::meczczyna },
                             { "Marianna", "Kowalska", 80, Plec::kobieta } };
13)     Lekarz rezydent = { "Tomasz", "Soltys", 30, Plec::meczczyna, nullptr };

14)     cout << pacjenci[0]
15)           << pacjenci[1];
16) }
```

KASKADOWE WYWOŁANIE  
OPERATORA <<

## WAŻNE:

- + w C++ można zdefiniować jak *cout* ma wypisywać nasz typ danych na ekranie,
- + bardzo ważny jest brak *const* przy referencji &str,
- + dzięki zwróceniu referencji przez operator<<, możliwe jest także wywołanie kaskadowe,
- + definiując działanie operatora<<, korzystamy ze zdefiniowanych już wcześniej operatorów<< dla innych typów danych.

C:\Windows\system32\cmd.exe

```
Pacjent:      Jan Nowak
               mezczyzna (45 lat).

Pacjent:      Marianna Kowalska
               kobieta (80 lat).
```

- Czym różni się struktura od tablicy?

Tablica to skończony zbiór elementów tego samego typu, rozróżnianych po indeksie. Struktura to zbiór elementów różnych typów, rozróżnianych po nazwach.

- Jaka są różnice między **enum**, a **enum class**?

Każdy **enum class** jest osobnym typem danych co pozwala mu na dublowanie stanów występujących w innych typach wyliczeniowych. Trzeba jednak definiować dla niego operatory. **Enum** definiuje stany globalnie, ale domyślnie zachowuje się jak typ **int**.

- Do czego służy operator `->`?

Do dostępu do składnika struktury, której instancja jest wskazywana wskaźnikiem.

- Jaka jest różnica między instancją a strukturą?

Struktura to definicja typu danych, instancja to faktycznie istniejący w pamięci egzemplarz tego typu.

- Dlaczego odradza się stosowanie makrodefinicji `#define` do definicji aliasów typów?

Gdyż `#define` jest instrukcją preprocesora, który tylko zastępuje fragmenty kodu. Nie pozwala to sprawdzać czy podstawienie jest poprawne, ani nie pozwala na redukcję złożoności definicji.

- Dlaczego polem struktury nie może być instancja tej struktury?

Gdyż oznaczało by to nieskończoną definicję rekurencyjną. Do budowy instancji potrzebna by była już taka sama instancja.

- Dlaczego polem struktury może być wskaźnik do tej struktury?

Gdyż wskaźnik to tylko informacja gdzie w pamięci znajduje się instancja struktury. Typ wskaźnika tylko informuje jak należy interpretować wskazywane dane.

- Jaka jest korzyść z dublowania klucza w typie wyliczeniowym?

Pozwala to tworzyć słowne aliasy wartości. Przykładowo `Miesiące::Styczeń = 1` i `Miesiące::Pierwszy = 1`, oznacza `Styczeń` i `Pierwszy` to dla komputera to to samo.

- Dlaczego do funkcji wypełniającej strukturę musimy przesłać instancję przez referencję lub wskaźnik?

Gdyż inaczej wypełnialibyśmy lokalną kopię instancji.

- Jak działa operator „.” (kropka) ?

Daje dostęp do pól struktury. Po lewej piszemy nazwę insynuacji struktury, po prawej nazwę pola.

- Jaka jest podstawowa rola struktur w programowaniu?

Łączenie logiczne danych różnych typów w nierozzerwalną paczkę.

- Czy można zdefiniować operator + dla pary typów: (int,int)?

Nie taki operator jest już zdefiniowany i nie można go przedefiniować.

## Dla przyszłych inżynierów (dla wszystkich):

1. Nauka kreatywnego rozwiązywania problemów w warunkach ograniczonego dostępu do narzędzi (programowanie to tylko kontekst – bogate źródło problemów do rozwiązywania 😊).
2. Nauka pracy z dokumentacją narzędzi informatycznych (umiejętność bardzo uniwersalna).

## Dla przyszłych programistów:

3. Nauka podstawowych technik programistycznych dostępnych w językach C i C++ i językach pochodnych.
4. Nauka sposobu myślenia stosownego w programowaniu proceduralnym i obiektowym (dotyczy wszystkich języków).
5. Nauka tworzenia aplikacji z graficznym interfejsem użytkownika (semestr III).
6. Nauka pracy z bibliotekami zewnętrznymi.

# PYTANIA?