Wykład 12: Typ strukturalny w języku C/C++.

dr inż. Andrzej Stafiniak

Wrocław 2023

Typy złożone – coś więcej

- Tworząc jakiekolwiek oprogramowanie mamy potrzebę operowania na różnego rodzaju informacjach – danych.
- W trakcie kursu poznaliśmy różne typy reprezentacji danych:
  - typy podstawowe (wybudowane) – np. `char, int, float,`
  - typy pochodne (złożone) – np. **tablice**…

Typ złożony jakim jest **tablica** stanowi przykład reprezentacji danych, gdzie za pomocą pojedynczego identyfikatora mamy możliwość operowania na wielu wartościach tego samego typu (ulokowanych w liniowym obszarze pamięci komputera).

Typy złożone – coś więcej

- Ale nie zawsze **tablica** będzie rozwiązaniem optymalny lub wystarczającym.

Jakie znamy jeszcze złożone typy danych?

Typy złożone – coś więcej

- Ale nie zawsze **tablica** będzie rozwiązaniem optymalny lub wystarczającym.

Jakie znamy jeszcze złożone typy danych?

Wyobraźmy sobie sytuację gdy istnieje potrzeba przechowania informacji różnego typu w jednym miejscu, aby za pomocą jednego identyfikatora mieć pogrupowane wszystkie potrzebne dane, np. informacje o studencie, dane teleadresowe, nr indeksu, oceny z kursów.

Typy złożone – coś więcej

- Ale nie zawsze **tablica** będzie rozwiązaniem optymalny lub wystarczającym.

Jakie znamy jeszcze złożone typy danych?

Wyobraźmy sobie sytuację gdy istnieje potrzeba przechowania informacji różnego typu w jednym miejscu, aby za pomocą jednego identyfikatora mieć pogrupowane wszystkie potrzebne dane, np. informacje o studencie, dane teleadresowe, nr indeksu, oceny z kursów.

W tym celu język C/C++ ma przewidziany specjalne typy danych:

- **struktura,**
- **unia.**

Typy złożone – struktura, unia

- **Struktura** – to obiekt danych grupujący pod jednym identyfikatorem wiele wartości różnych typów (podstawowych i pochodnych). Może przechowywać jednocześnie kilka typów danych, ułożonych kolejno w liniowym obszarze pamięci.

- **Unia** - to również obiekt danych grupujący pod jednym identyfikatorem wiele wartości różnych typów, **ALE** różni się od **struktury** tym, że w danej chwili może operować na jednym elemencie składowym. Wszystkie składniki/pola **unii** zajmują ten sam obszar pamięci, o rozmiarze największego z nich.

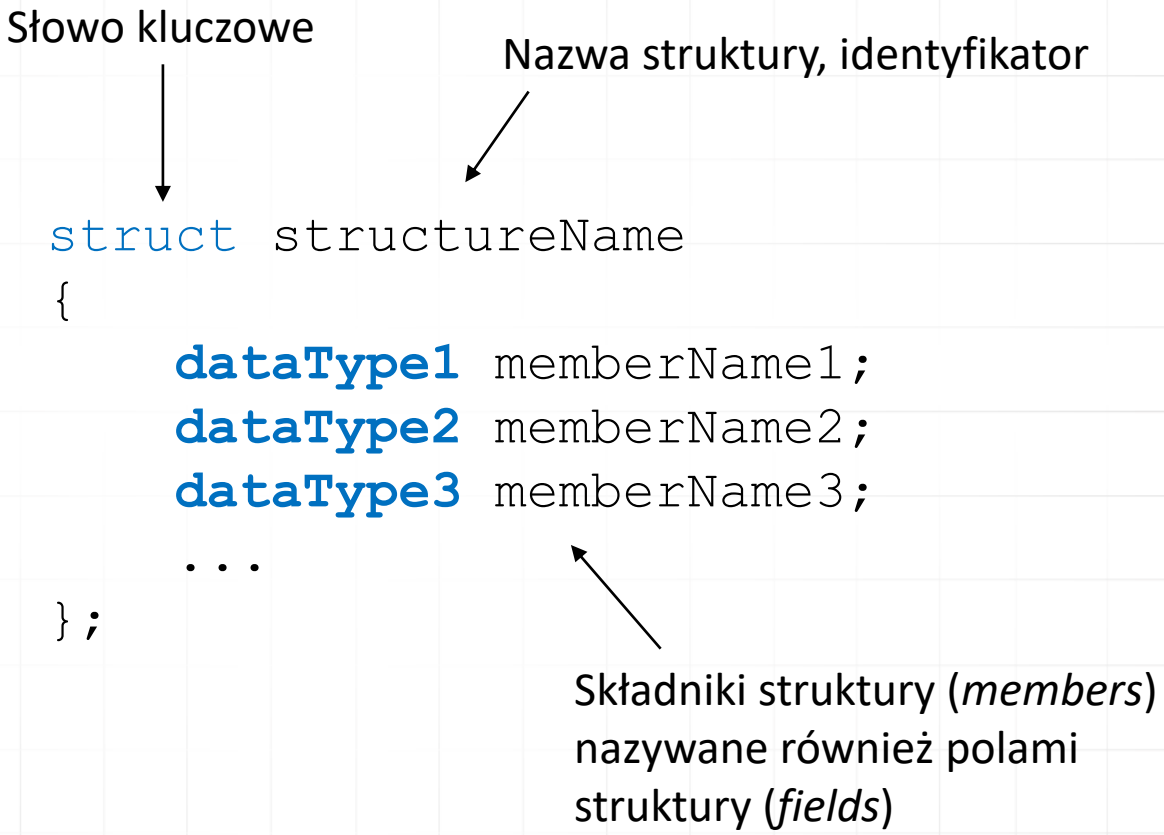
Typ strukturalny

Struktura to złożony typ danych, pozwalająca przechowywać różne informacje. Dzięki strukturom w łatwy sposób można tworzyć zbiory / bazy danych.

Struktury to pierwszy krok w kierunku **klas** w języku C++, a wiec w kierunku paradygmatu programowania obiektowego.

Deklaracja struktury

Składnia:

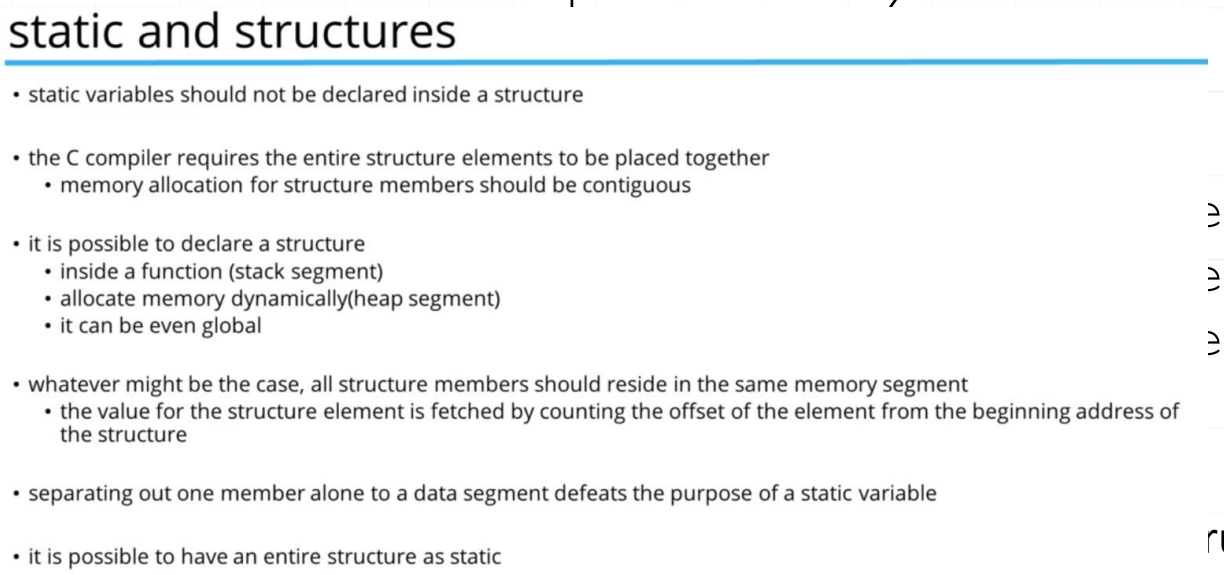


Deklaracja struktury

Składnia:

Słowo kluczowe

Nazwa struktury, identyfikator




```
e1;
e2;
e3;
```

Składnik
dane re
pomocą
- typó
- typó
  - tablic,
  - innych struktur.

ruktury (*members*)
nazywane również polami
struktury (*fields*)

Deklaracja struktury

Przykład:

```
struct Student
{
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
};
```

- Struktura Student została zdeklarowana z wykorzystaniem słowa kluczowego `struct`.
- Po podaniu nazwy struktury wewnątrz nawiasów klamrowych umieszcza się / deklaruje składowe / pola struktury.
- Deklaracja struktury zakończona jest średnikiem.

Deklaracja struktury, instancja oraz inicjalizacja

Przykład:

```
#include<stdio.h>

struct Student
{
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
}studentW5, studentW11;

int main(){

    struct Student studentW12n = {
    "Jan",
    "Kowalski",
    21,
    266666
    };
    return 0;
}
```

- Stworzenie instancji (egzemplarza, obiektu) struktury Student o nazwie `studentW5`, `studentW11` podczas deklaracji.
- W celu instancjonowania zmiennej strukturalnej w innym miejscu można użyć ponownie słowa kluczowego `struct`, nazwy struktury oraz nazwy zmiennej.
- W języku C++ nazwa struktury jest pełnoprawnym typem zmiennej i nie trzeba stosować słowa `struct`.

  `Student studentW12n;`
- Wykorzystując listę inicjalizacyjną istnieje możliwość wypełnienia kolejnych pól zmiennej strukturalnej `studentW12n`, każda ze składowych oddzielona przecinkiem.

Deklaracja struktury, instancja oraz inicjalizacja

Przykład:

- W języku C++ nazwa struktury jest pełnoprawnym typem zmiennej i nie trzeba stosować słowa `struct`.
- W języku C można posłużyć się **mechanizmem aliasów typów** aby pominąć słowo `struct`. Realizowane jest to za pośrednictwem słowa kluczowego `typedef` (definiowanie własnego typu złożonego).

```
#include<stdio.h>

struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
};

int main(){

    typedef struct Student StudentPwr;

    StudentPwr studentW12n = {
        ”Jan”,
        ”Kowalski”,
        21,
        266666
    };
    return 0;
}
```

```
#include<stdio.h>

typedef struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index
}StudentPwr;

int main(){

    StudentPwr studentW12n = {
        ”Jan”,
        ”Kowalski”,
        21,
        266666
    };
}
```

W tej sytuacji to nie instancja, a **alias typu** złożonego na bazie struktury Student

Trochę o typedef

Typedef vs. #define:

```
#include<stdio.h>

struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
};

int main(){

    typedef struct Student StudentPwr;

    StudentPwr studentW12n = {
        "Jan",
        "Kowalski",
        21,
        266666
    };
    return 0;
}
```

W tej sytuacji to nie instancja, a **alias typu** złożonego na bazie struktury Student

Deklaracja struktury, instancja oraz inicjalizacja

Przykład:

```
#include<iostream>

struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;

};

int main(){

    using StudentPwr = Student;

    StudentPwr studentW12n = {
        ”Jan”,
        ”Kowalski”,
        21,
        266666
    };
    return 0;
}
```

Trochę więcej o mechanizmie aliasów typów:

- W języku C++ wprowadzono również możliwość posługiwania się **mechanizmem aliasów typów** za pomocą słowa kluczowego `using`.
- `StudentPwr` jest aliasem typu struktury `Student`.
- `studentW12n` – jest zainicjalizowaną instancją/egzemplarzem struktury `Student`.
- Z faktu że C++ jest wstecznie kompatybilny, działa tu również mechanizm z `typedef`.

Zagnieżdżanie struktur

Przykład:

```
#include<stdio.h>
typedef struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
    struct {
        float matematyka;
        float fizyka;
    }ocena;
}StudentPwr;

int main(){
    StudentPwr studentW12n = {
        "Jan",
        "Kowalski",
        21,
        266666
        {  4.5,
           5.0  }
    };
    return 0;
}
```

- Istnieje możliwość tworzenie nienazwanych typów struktur, **struktur anonimowych**.
- Mogą one posiadać tylko jedną instancję, utworzoną w miejscu deklaracji struktury
- **Struktury anonimowe** najczęściej wykorzystywane są do **zagnieżdżania** w innych strukturach, a ich instancje stają się polem tych struktur.

Odwołanie się do pola struktury

Przykład:

```
#include<stdio.h>
typedef struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
    struct {
        float matematyka;
        float fizyka;
    }ocena;
}StudentPwr;

int main(){
    StudentPwr studentW12n = {
        "Jan",
        "Kowalski",
        21,
        266666
        {  4.5,
           5.0  }
    };
    return 0;
}
```

- Aby odwołać się do **pola struktury** (zaczytać z, zapisać do) korzysta się z **operatora dostępu do składowych** – operatora **.**

```
printf("Student %s %s uzyskal %f z fizyki",
                    studentW12n.imie,
                    studentW12n.nazwisko,
                    studentW12n.ocena.fizyka);

studentW12n.ocena.matematyka = 5.0;
```

Tablica struktur

Przykład:

```
#include<stdio.h>
typedef struct Student {
    char imie[20];
    char nazwisko[50];
    short wiek;
    int index;
    struct {
        float matematyka;
        float fizyka;
    }ocena;
}StudentPwr;
int main(){
    StudentPwr studentW12n[99];
    for (int i=0; i<99; ++i){
        printf("Wprowadz dane %d-ego studenta:", i+1)
        printf("\nImie:");
        scanf("%s", studentW12n[i].imie);
        printf("\nNazwisko:");
        scanf("%s", studentW12n[i].nazwisko);
    }
    return 0;
}
```

- W ten sam sposób, jak deklaruje się tablicę elementów typu prostego, tak samo można stworzyć tablicę struktur.
- Aby odwołać się do elementu tablicy typu struktura `StudentW12n` należy podać indeks obiektu w nawiasach [].

```
for (int i=0; i<99; ++i){
  printf("Student %s %s", studentW12n[i].imie,
                          studentW12n[i].nazwisko)
 }
}
```

Struktura jako argument funkcji

- Struktury mogą być przekazywane do funkcji, jak i stanowić wartość zwracaną z funkcji.

```c
#include <stdio.h>
#include <math.h>

typedef struct Point {
    double x;
    double y;
} Point_t;

double calculateDistance ( Point_t first , Point_t second ) {
    const double xDist = second.x - first.x;
    const double yDist = second.y - first.y;
    return sqrt ( xDist*xDist + yDist*yDist );
}

Point_t movePoint(Point_t point){
    point.x = point.x + 10;
    point.y = point.y + 10;
    return point;
}

int main () {
    Point_t first = {1.0 , 2.0};
    Point_t second = {3.0 , 5.0};

    printf ("Distance between two points is equal %f", calculateDistance(first , second));

    first = movePoint(first);

    printf ("\nCurrent distance between two points is equal %f", calculateDistance(first , second));

    return 0;
}
```

Zmodyfikowany przykład z instrukcji

```
Distance between two points is equal 3.605551
Current distance between two points is equal 10.630146
```

Wskaźnik na strukturę

- Możliwość stosowania wskaźników na strukturę pozwala na dynamiczne tworzenie obiektów tego typu.
- Posługując się wskaźnikami należy pamiętać, że wskazujemy adres obiektu, dlatego aby odwoływać się do składowych pól struktury należy posługiwać się operatorem wyłuskania/dereferencji ***** i operatora **.** , ale również mamy możliwość posługiwania się operatorem dostępu do składowej przez wskaźnik **->**.

Wskaźnik na strukturę

```c
#include <stdio.h>
#include <stdlib.h> // calloc(), free()

typedef struct Point {
    int x;
    int y;
} Point_t;


int main () {
    unsigned int size;
    printf("Wprowadz ilosc punktow:");
    scanf("%d", &size);

    Point_t * first = (Point_t *)calloc(size, sizeof(Point_t));

    for(int i=0; i<size; ++i){
        (first+i)->x = 1;
        (first+i)->y = 1;
    }

    for(int i=0; i<size; ++i){
        printf ("\nWsp. punktu nr %d - %d, %d", i+1, (first+i)->x, first[i].y);
    }

    for(int i=0; i<size; ++i){
        printf ("\nPodaj aktualne wsp. punktu nr %d (x i y):\n", i+1);
        scanf(" %d", &((first+i)->x));
        scanf(" %d", &(first[i].y));
    }

    for(int i=0; i<size; ++i){
        printf ("\nAktualne wsp. punktu nr %d - %d, %d", i+1, first[i].x, (*(first+i)).y);
    }

        free(first);

    return 0;
}
```

Przykład, w którym wykorzystano wskaźnik na strukturę w celu stworzenie dynamicznej tablicy struktur. Wykorzystano różne sposoby dostępu do składowych struktury.

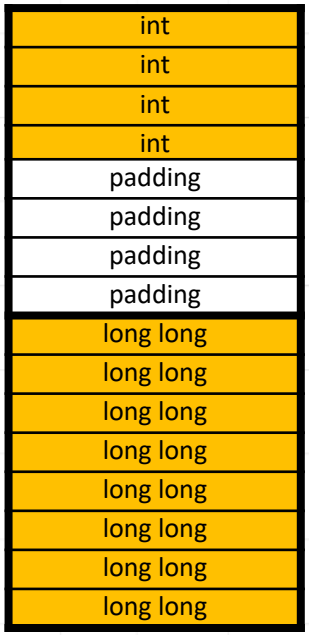
```
Wprowadz ilosc punktow:1

Wsp. punktu nr 1 - 1, 1
Podaj aktualne wsp. punktu nr 1 (x i y):
3
4

Aktualne wsp. punktu nr 1 - 3, 4
```

Rozmiar składowych struktury - padding

```
struct {
   int x;
   long long y;
}point;
```

- Gdy sprawdzimy rozmiar struktury oraz rozmiar każdego pola danej struktury może okazać się że suma rozmiarów pół będzie różna od rozmiaru całej struktury.
- W strukturach kolejne pola w pamięci ułożone są w sposób ciągły, ale do reprezentacji całej struktury każde pole może być **wyrównane** do rozmiaru **największego pola struktury**. **Wyrównanie, dopełnienie** komórkami pamięci do rozmiaru największego pola nazywamy ***paddingiem.***
- Ma to uzasadnienie ze względu na optymalizację wykonywania kodu. W czasie jednego cyklu procesor operuje na porcji danych zwanej **słowem maszynowym**.
- **Słowo maszynowe** w architekturze 32b wynosi 4B, a w 64b – 8B.

64 bit:

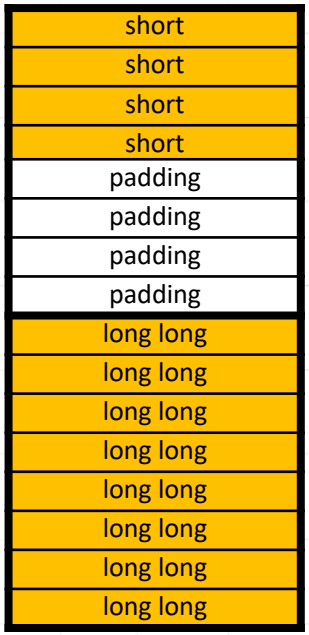


Rozmiar składowych struktury - padding

- **Dopełnienia** mają za zadanie optymalizacji, aby w miarę możliwości procesor mógł, w jednej porcji (cyklu), przyjąć całą wartość zapisaną w konkretnym polu struktury (ewentualnie kilka wartości w całości)
- **Dopełnienie** nie jest realizowane między elementami tablicy.
- Standard języka nie definiuje w jaki sposób realizować dopełnienia. Będzie to zależne od architektury, kompilatora.

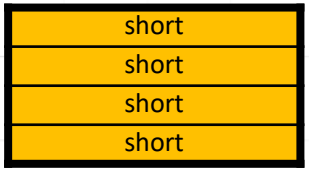
```
struct {
   short x;
   short y;
   long long z;
}point;
```

64 bit:



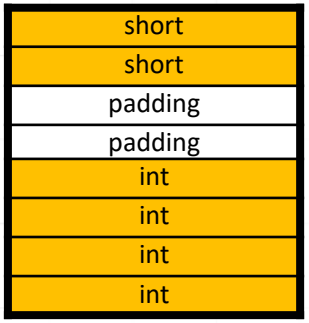
Organizacja danych w strukturze 64 bit:

```
struct {
  short x;
  short y;
}point;
```



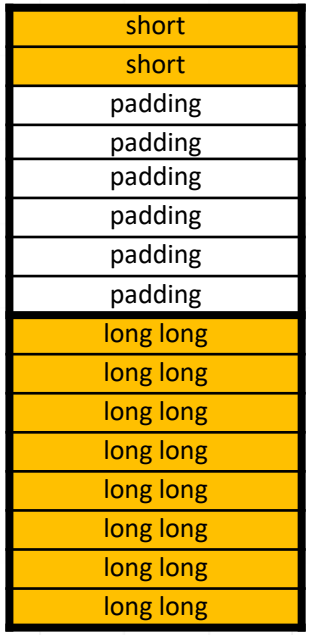
rozmiar point - 4B
padding - 0B

```
struct {
  short x;
  int y;
}point;
```



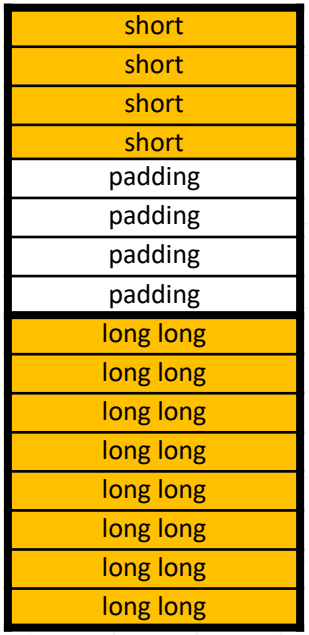
rozmiar point - 8B
padding - 2B

```
struct {
  short x;
  long long y;
}point;
```



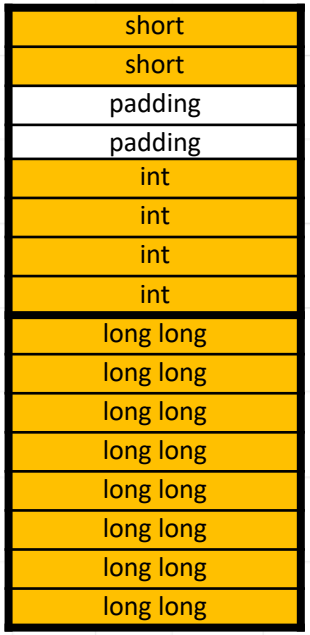
rozmiar point - 16B
padding - 6B

```
struct {
  short x;
  short y;
  long long z;
}point;
```



rozmiar point - 16B
padding - 4B

```
struct {
  short x;
  int y;
  long long z;
}point;
```

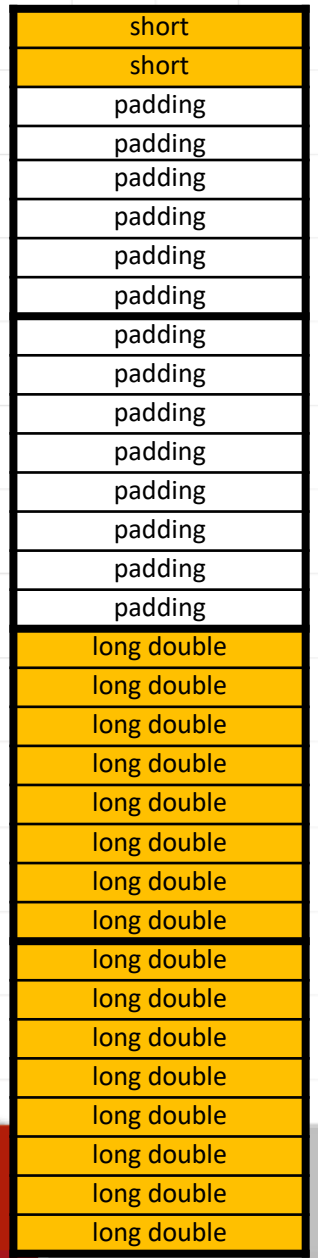


rozmiar point - 16B
padding - 2B

Organizacja danych w strukturze

64 bit:

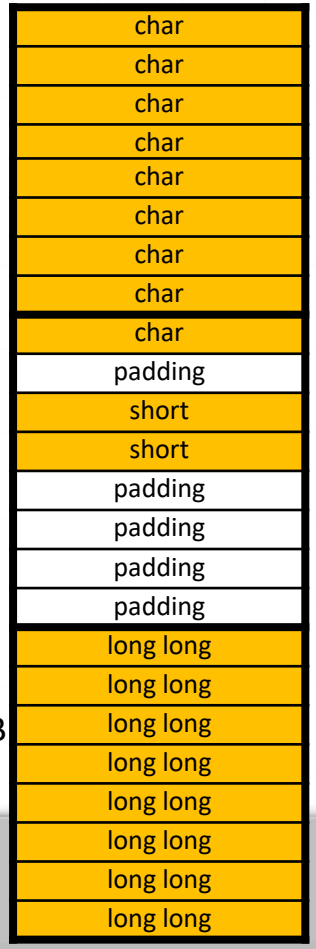
```
struct {
  short x;
  long double y;
}point;
```



rozmiar point - 32B
padding - 14B

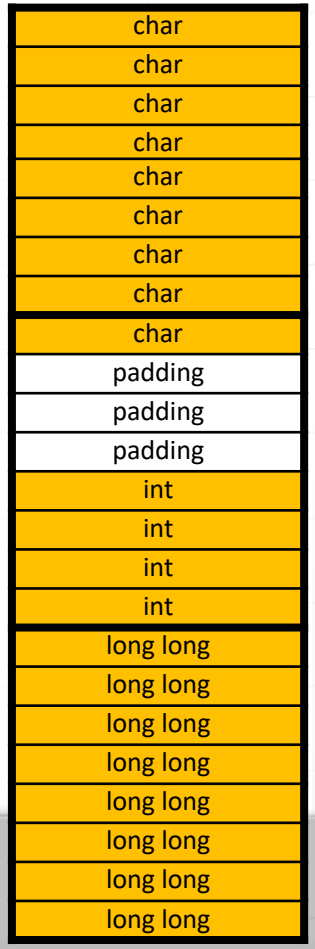
(MinGW – 32bit:
rozmiar point - 16B
2B+2B+12B)

```
struct {
  char name[9];
  short y;
  long long z;
}point;
```

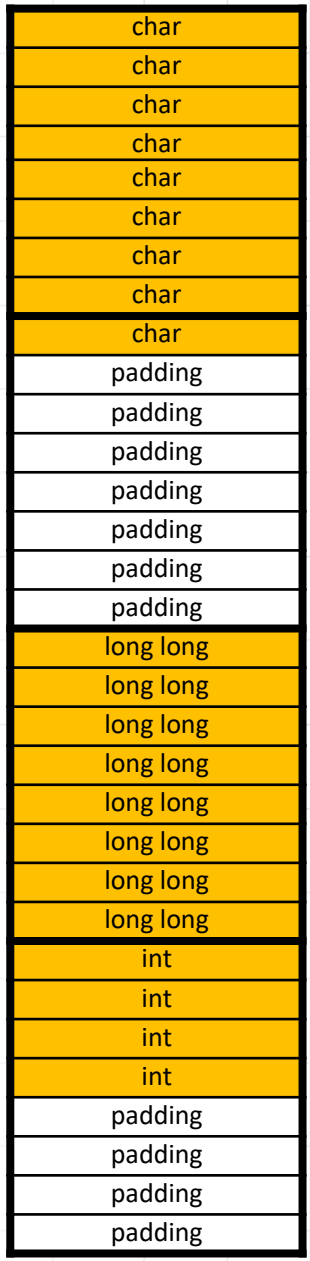


rozmiar point - 24B
padding - 5B

```
struct {
  char name[9];
  int y;
  long long z;
}point;
```




```
struct {
  char name[9];
  long long z;
  int y;
}point;
```



rozmiar point - 24B
padding - 3B

Rozmiar składowych struktury - padding

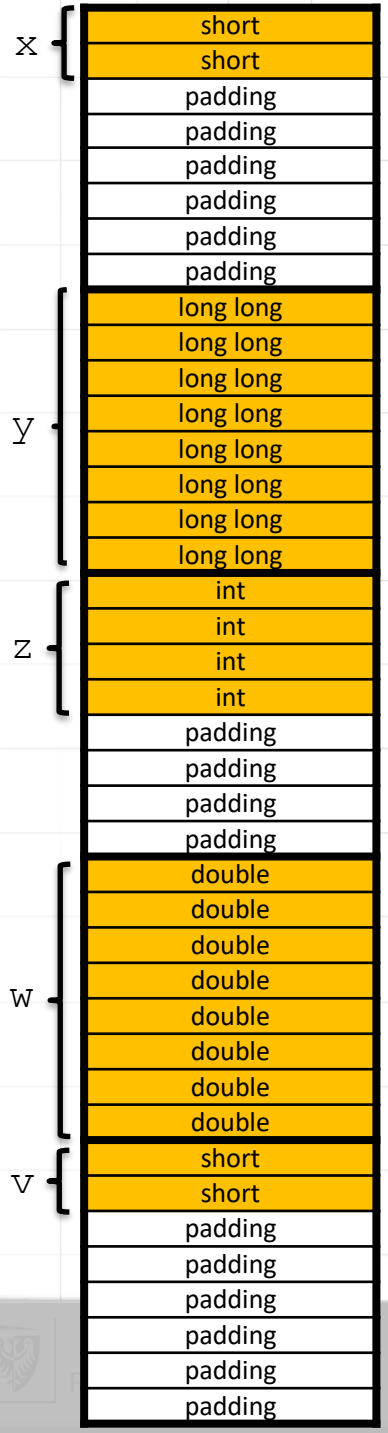
- Ponieważ w strukturach kolejne **pola** w pamięci ułożone są w sposób **ciągły**, zgodnie z kolejnością deklaracji oraz pola te są **dopełniane** do rozmiarów największego z **pól**, odpowiednie ich ułożenie może prowadzić do **optymalizacji rozmiaru** pamięci zajmowanej przez **instancję** tej struktury.

(Zmodyfikowany przykład z dodatku w instrukcjach.)

```c
#include <stdio.h>

struct MessyStruct {
    short x;
    long long y;
    int z;
    double w;
    short v;
} messyStruct;

int main() {
    int xyzwv= sizeof messyStruct.x
    + sizeof messyStruct.y + sizeof messyStruct.z
    + sizeof messyStruct.w + sizeof messyStruct.v;

    printf("Rozmiar messyStruct: %d\n", sizeof messyStruct);
    printf("Rozmiar pola x: %d\n",sizeof messyStruct.x);
    printf("Rozmiar pola y: %d\n",sizeof messyStruct.y);
    printf("Rozmiar pola z: %d\n",sizeof messyStruct.z);
    printf("Rozmiar pola w: %d\n",sizeof messyStruct.w);
    printf("Rozmiar pola v: %d\n",sizeof messyStruct.v);
    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola x: %d\n",&messyStruct.x);
    printf("Adres pola y: %d\n",&messyStruct.y);
    printf("Adres pola z: %d\n",& messyStruct.z);
    printf("Adres pola w: %d\n",&messyStruct.w);
    printf("Adres pola v: %d",&messyStruct.v);

    return 0;
}
```

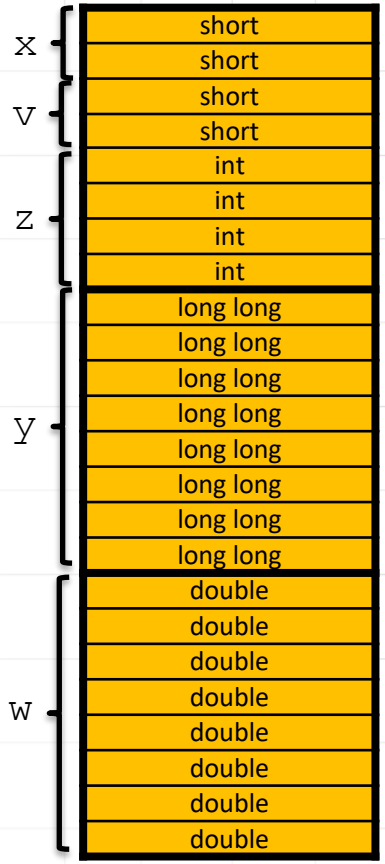
```
Rozmiar messyStruct: 40
Rozmiar pola x: 2
Rozmiar pola y: 8
Rozmiar pola z: 4
Rozmiar pola w: 8
Rozmiar pola v: 2
Rozmiar sumy pol: 24

Adres pola x: 4223104
Adres pola y: 4223112
Adres pola z: 4223120
Adres pola w: 4223128
Adres pola v: 4223136
```

padding - 16B



Rozmiar składowych struktury - padding

- Ponieważ w strukturach kolejne **pola** w pamięci ułożone są w sposób **ciągły**, zgodnie z kolejnością deklaracji oraz pola te są **dopełniane** do rozmiarów największego z **pól**, odpowiednie ich ułożenie może prowadzić do **optymalizacji rozmiaru** pamięci zajmowanej przez **instancję** tej struktury.



(Zmodyfikowany przykład z dodatku w instrukcjach.)

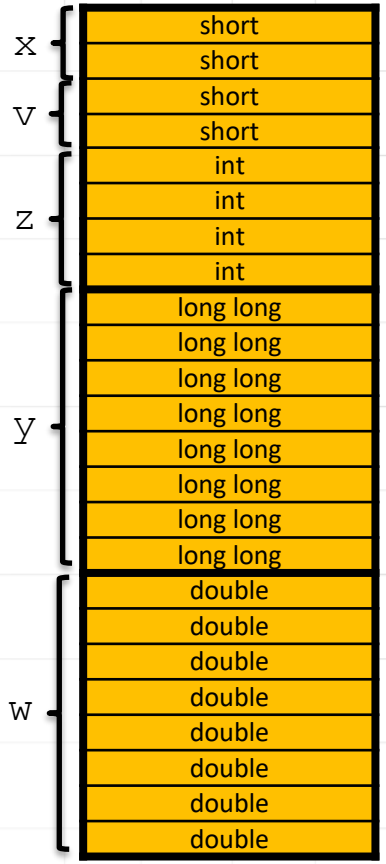
```c
#include <stdio.h>

struct OrderlyStruct {
    short x;
    short v;
    int z;
    long long y;
    double w;
} orderlyStruct;

int main() {
    int xyzwv= sizeof orderlyStruct.x
    + sizeof orderlyStruct.y + sizeof orderlyStruct.z
    + sizeof orderlyStruct.w + sizeof orderlyStruct.v;

    printf("Rozmiar orderlyStruct: %d\n", sizeof orderlyStruct);
    printf("Rozmiar pola x: %d\n",sizeof orderlyStruct.x);
    printf("Rozmiar pola v: %d\n",sizeof orderlyStruct.v);
    printf("Rozmiar pola z: %d\n",sizeof orderlyStruct.z);
    printf("Rozmiar pola y: %d\n",sizeof orderlyStruct.y);
    printf("Rozmiar pola w: %d\n",sizeof orderlyStruct.w);

    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola x: %d\n",&orderlyStruct.x);
    printf("Adres pola y: %d\n",&orderlyStruct.v);
    printf("Adres pola z: %d\n",&orderlyStruct.z);
    printf("Adres pola w: %d\n",&orderlyStruct.y);
    printf("Adres pola v: %d",&orderlyStruct.w);

    return 0;
}
```

padding - 0B

```
Rozmiar orderlyStruct: 24
Rozmiar pola x: 2
Rozmiar pola v: 2
Rozmiar pola z: 4
Rozmiar pola y: 8
Rozmiar pola w: 8
Rozmiar sumy pol: 24

Adres pola x: 4223088
Adres pola y: 4223090
Adres pola z: 4223092
Adres pola w: 4223096
Adres pola v: 4223104
```

Rozmiar składowych struktury - padding

- Ponieważ w strukturach kolejne **pola** w pamięci ułożone są w sposób **ciągły**, zgodnie z kolejnością deklaracji oraz pola te są **dopełniane** do rozmiarów największego z **pól**, odpowiednie ich ułożenie może prowadzić do **optymalizacji rozmiaru** pamięci zajmowanej przez **instancję** tej struktury.



- Aby skutecznie optymalizować rozmiar, należy deklarować pola tak aby tworzyły monotoniczny ciąg tych samych typów.

```c
#include <stdio.h>

struct OrderlyStruct {
    short x;
    short v;
    int z;
    long long y;
    double w;
} orderlyStruct;

int main() {
    int xyzwv= sizeof orderlyStruct.x
    + sizeof orderlyStruct.y + sizeof orderlyStruct.z
    + sizeof orderlyStruct.w + sizeof orderlyStruct.v;

    printf("Rozmiar orderlyStruct: %d\n", sizeof orderlyStruct);
    printf("Rozmiar pola x: %d\n",sizeof orderlyStruct.x);
    printf("Rozmiar pola v: %d\n",sizeof orderlyStruct.v);
    printf("Rozmiar pola z: %d\n",sizeof orderlyStruct.z);
    printf("Rozmiar pola y: %d\n",sizeof orderlyStruct.y);
    printf("Rozmiar pola w: %d\n",sizeof orderlyStruct.w);

    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola x: %d\n",&orderlyStruct.x);
    printf("Adres pola y: %d\n",&orderlyStruct.v);
    printf("Adres pola z: %d\n",&orderlyStruct.z);
    printf("Adres pola w: %d\n",&orderlyStruct.y);
    printf("Adres pola v: %d",&orderlyStruct.w);

    return 0;
}
```

padding - 0B

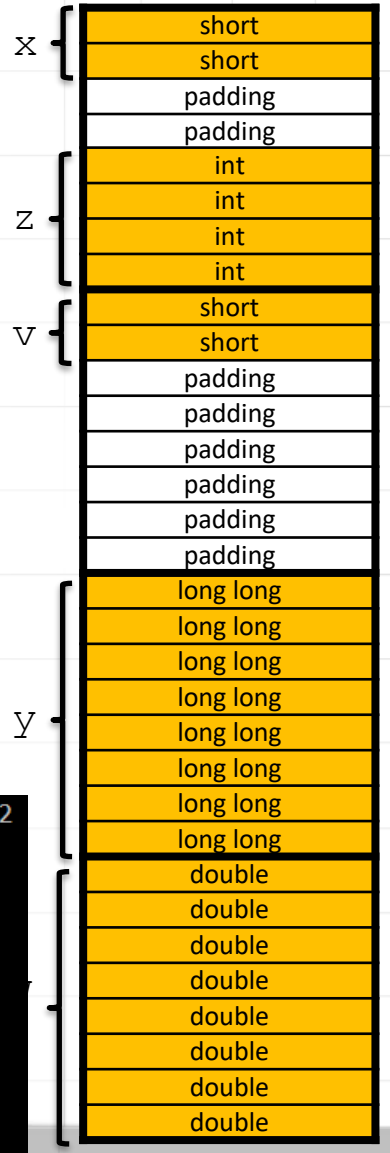
```
Rozmiar orderlyStruct: 24
Rozmiar pola x: 2
Rozmiar pola v: 2
Rozmiar pola z: 4
Rozmiar pola y: 8
Rozmiar pola w: 8
Rozmiar sumy pol: 24

Adres pola x: 4223088
Adres pola y: 4223090
Adres pola z: 4223092
Adres pola w: 4223096
Adres pola v: 4223104
```

Rozmiar składowych struktury - padding

- Aby skutecznie optymalizować rozmiar, należy deklarować pola tak aby tworzyły ciąg tych samych typów.
- To znaczy starać się grupować pola tego samego typu w obrębie rozmiaru najdłuższego pola.

```c
#include <stdio.h>

struct AlmostOrderlyStruct {
    short x;
    int z;
    short v;
    long long y;
    double w;
} almostOrderlyStruct;

int main() {
    int xyzwv= sizeof almostOrderlyStruct.x
    + sizeof almostOrderlyStruct.y + sizeof almostOrderlyStruct.z
    + sizeof almostOrderlyStruct.w + sizeof almostOrderlyStruct.v;

    printf("Rozmiar almostOrderlyStruct: %d\n", sizeof almostOrderlyStruct);
    printf("Rozmiar pola x: %d\n",sizeof almostOrderlyStruct.x);
    printf("Rozmiar pola z: %d\n",sizeof almostOrderlyStruct.z);
    printf("Rozmiar pola v: %d\n",sizeof almostOrderlyStruct.v);
    printf("Rozmiar pola y: %d\n",sizeof almostOrderlyStruct.y);
    printf("Rozmiar pola w: %d\n",sizeof almostOrderlyStruct.w);

    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola x: %d\n",&almostOrderlyStruct.x);
    printf("Adres pola z: %d\n",&almostOrderlyStruct.z);
    printf("Adres pola y: %d\n",&almostOrderlyStruct.v);
    printf("Adres pola w: %d\n",&almostOrderlyStruct.y);
    printf("Adres pola v: %d",&almostOrderlyStruct.w);

    return 0;
}
```

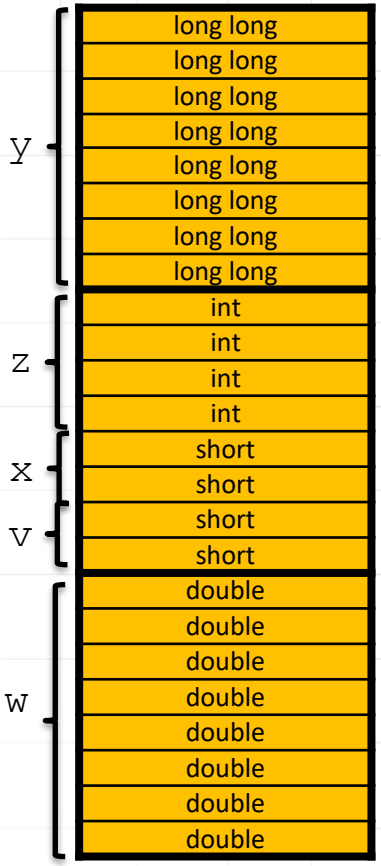
```
Rozmiar almostOrderlyStruct: 32
Rozmiar pola x: 2
Rozmiar pola z: 4
Rozmiar pola v: 2
Rozmiar pola y: 8
Rozmiar pola w: 8
Rozmiar sumy pol: 24

Adres pola x: 4223104
Adres pola z: 4223108
Adres pola y: 4223112
Adres pola w: 4223120
Adres pola v: 4223128
```



padding - 8B

Rozmiar składowych struktury - padding

- Aby skutecznie optymalizować rozmiar, należy deklarować pola tak aby tworzyły ciąg tych samych typów.
- To znaczy starać się grupować pola tego samego typu w obrębie rozmiaru najdłuższego pola.



padding - 0B

```c
#include <stdio.h>

struct OrderedStruct {
    long long y;
    int z;
    short x;
    short v;
    double w;

} orderedStruct;

int main() {
    int xyzwv= sizeof orderedStruct.x
    + sizeof orderedStruct.y + sizeof orderedStruct.z
    + sizeof orderedStruct.w + sizeof orderedStruct.v;

    printf("Rozmiar orderedStruct: %d\n", sizeof orderedStruct);
    printf("Rozmiar pola y: %d\n",sizeof orderedStruct.y);
    printf("Rozmiar pola z: %d\n",sizeof orderedStruct.z);
    printf("Rozmiar pola x: %d\n",sizeof orderedStruct.x);
    printf("Rozmiar pola v: %d\n",sizeof orderedStruct.v);
    printf("Rozmiar pola w: %d\n",sizeof orderedStruct.w);
    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola y: %d\n",&orderedStruct.y);
    printf("Adres pola z: %d\n",&orderedStruct.z);
    printf("Adres pola x: %d\n",&orderedStruct.x);
    printf("Adres pola v: %d\n",&orderedStruct.v);
    printf("Adres pola w: %d\n",&orderedStruct.w);

    return 0;
}
```

```
Rozmiar orderedStruct: 24
Rozmiar pola y: 8
Rozmiar pola z: 4
Rozmiar pola x: 2
Rozmiar pola v: 2
Rozmiar pola w: 8
Rozmiar sumy pol: 24

Adres pola y: 4223088
Adres pola z: 4223096
Adres pola x: 4223100
Adres pola v: 4223102
Adres pola w: 4223104
```

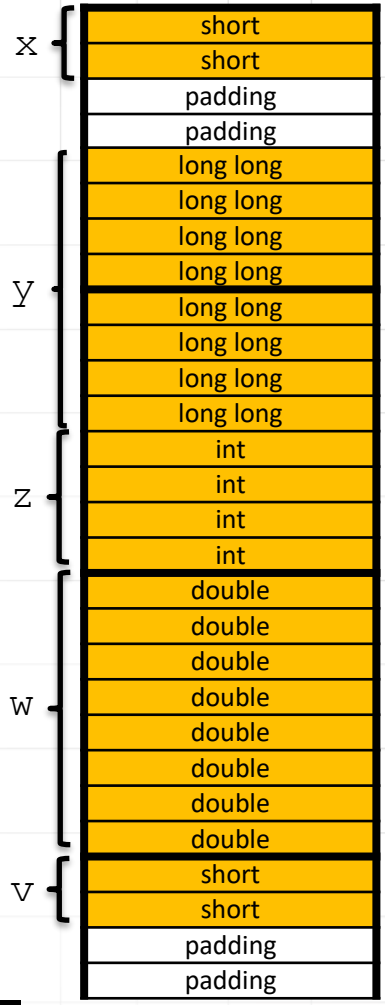
Rozmiar składowych struktury - padding

- Dyrektywa `#pragma pack` – maksymalna długość **dopełnienia,** poprzez ustawienie minimalnego rozmiaru pola. Ustawiany rozmiar pola musi być potęgą liczby 2.

- Dyrektywa nie ustawia rozmiaru **pola** większego niż **najdłuższe pole**.

- Istnieje jeszcze możliwość redukcji dopełnienia za pomocą **atrybutu packed** (instrukcja 9).

- Oba sposoby łączą się ze spadkiem wydajności wykonywania programu.




```c
#include <stdio.h>

#pragma pack (4)

struct AlmostOrderlyStruct {
    short x;
    long long y;
    int z;
    double w;
    short v;
} almostOrderlyStruct;

int main() {
    int xyzwv= sizeof almostOrderlyStruct.x
    + sizeof almostOrderlyStruct.y + sizeof almostOrderlyStruct.z
    + sizeof almostOrderlyStruct.w + sizeof almostOrderlyStruct.v;

    printf("Rozmiar almostOrderlyStruct: %d\n", sizeof almostOrderlyStruct);
    printf("Rozmiar pola x: %d\n",sizeof almostOrderlyStruct.x);
    printf("Rozmiar pola y: %d\n",sizeof almostOrderlyStruct.y);
    printf("Rozmiar pola z: %d\n",sizeof almostOrderlyStruct.z);
    printf("Rozmiar pola w: %d\n",sizeof almostOrderlyStruct.w);
    printf("Rozmiar pola v: %d\n",sizeof almostOrderlyStruct.v);
    printf("Rozmiar sumy pol: %d\n\n",xyzwv);

    printf("Adres pola x: %d\n",&almostOrderlyStruct.x);
    printf("Adres pola w: %d\n",&almostOrderlyStruct.y);
    printf("Adres pola z: %d\n",&almostOrderlyStruct.z);
    printf("Adres pola v: %d\n",&almostOrderlyStruct.w);
    printf("Adres pola y: %d\n",&almostOrderlyStruct.v);

    return 0;
}
```

```
Rozmiar almostOrderlyStruct: 28
Rozmiar pola x: 2
Rozmiar pola y: 8
Rozmiar pola z: 4
Rozmiar pola w: 8
Rozmiar pola v: 2
Rozmiar sumy pol: 24

Adres pola x: 4223088
Adres pola w: 4223092
Adres pola z: 4223100
Adres pola v: 4223104
Adres pola y: 4223112
```

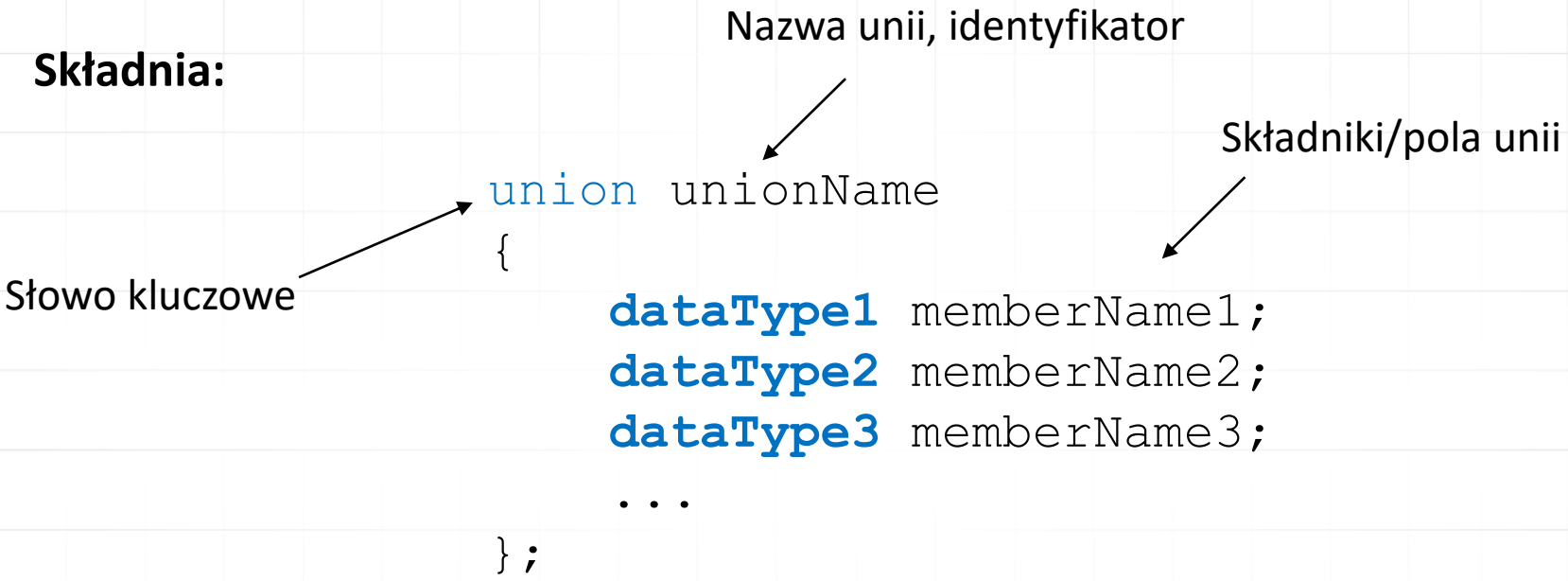
M. Stępniak, Laboratorium Informatyki

Typ złożony – unia

- W przypadku **unii -** jeden fragment pamięci może być interpretowany na kilka sposobów, a rozmiar **obiektu** będzie określony przez największy z jej składników.
- Inaczej, wszystkie składniki/pola **unii** zajmują ten sam obszar pamięci, o rozmiarze największego z nich. W danej chwili (instancji) możemy operować na jednym elemencie składowym **unii**.
- **Unie** stosuje się w celu oszczędności pamięci, np. w programowaniu mikrokontrolerów, w sytuacji gdy np. funkcja operuje na różnych typach danych, ale tylko na jednego typie z nich w danym czasie**.**

Typ złożony – unia

Składnia:



Union

- Zasady definiowania **unii**, wykorzystywania **aliasów typów**, **instancjonowania** podobnie jak w strukturach.
- Zastosowanie **unii** – instrukcja 9, dodatek str. 46-49.

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

union Number{
    int valueI;
    long long valueLL;
    float valueF;
    double valueD;
    long double valueLD;
}someNumber; // instancja unni, obiekt typu Number

int main() {
    cout << "Rozmiar unni Number:" << endl;
    cout << "   sizeof(Number) = " << sizeof(Number) << endl;
    cout << "Rozmiar skladowych unni Number:" << endl;
    cout << "   sizeof(int) = " << sizeof(int) << endl;
    cout << "   sizeof(long long) = " << sizeof(long long) << endl;
    cout << "   sizeof(float) = " << sizeof(float) << endl;
    cout << "   sizeof(double) = " << sizeof(double) << endl;
    cout << "   sizeof(long double) = " << sizeof(long double) << endl << endl;

    union Number anotherNumber; // instancja unni, obiekt typu Number

    someNumber.valueI = 11;
    someNumber.valueD = 11.1;

    anotherNumber.valueLL = 22ll;
    anotherNumber.valueLD = 22.2L;

    // obiekty typu Number (unnie) sa w satnie przechowywać tylko jeden element
    cout << "someNumber.valueI = " << someNumber.valueI << endl;
    cout << "someNumber.valueD = " << someNumber.valueD << endl << endl;

    cout << "adres someNumber.valueI = " << &someNumber.valueI << endl;
    cout << "adres someNumber.valueD = " << &someNumber.valueD << endl << endl;

    cout << "anotherNumber.valueLL = " << anotherNumber.valueLL << endl;
    cout << "anotherNumber.valueLD = " << anotherNumber.valueLD << endl << endl;

    cout << "adres anotherNumber.valueLL = " << &anotherNumber.valueLL << endl;
    cout << "adres anotherNumber.valueLD = " << &anotherNumber.valueLD << endl;

    return 0;
}
```

```
Rozmiar unni Number:
        sizeof(Number) = 16
Rozmiar skladowych unni Number:
        sizeof(int) = 4
        sizeof(long long) = 8
        sizeof(float) = 4
        sizeof(double) = 8
        sizeof(long double) = 12

someNumber.valueI = 858993459
someNumber.valueD = 11.1

adres someNumber.valueI = 0x408020
adres someNumber.valueD = 0x408020

anotherNumber.valueLL = -5649315372573550182
anotherNumber.valueLD = 22.2

adres anotherNumber.valueLL = 0x61fef0
adres anotherNumber.valueLD = 0x61fef0
```