5 Złożone typy danych

Pod pojęciem *złożonego typu danych* rozumiemy taki typ, którego wartości składają się z innych wartości. Możemy zatem w obiekcie typu złożonego wydzielić jego składowe innych typów.

W rozdziale tym zostaną omówione takie złożone typy danych jak:

- tablice
- struktury (rekordy)
- unie

Szczególnym przypadkiem złożonych danych są struktury dynamiczne, które mogą rosnąć i zmniejszać się podczas działania programu. Zostaną one omówione w rozdziale 8.

5.1 Tablice

Rozpatrzmy problem zsumowania 10 000 wartości całkowitych. Gdyby były one przechowywane w prostych zmiennych całkowitych, to wyrażenie obliczające ich sumę byłoby bardzo długie.

Jeszcze gorsza sytuacja byłaby w przypadku potrzeby przetwarzania n liczb. Bez odpowiedniej notacji nie byłoby to możliwe w programie, który ma ustaloną (niezmienną) długość.

Podstawowym typem złożonym jest tablica. Umożliwia ona zwięzły zapis kodu operującego na dużych danych.

Jesteśmy przyzwyczajenie do operowania we wzorach matematycznych indeksowanymi ciągami (a_i) oraz tablicami (a_{ij}). Podobną notację mamy w języku C.

Tablicę złożoną z n elementów tego samego typu T deklaruje się w następujący sposób:

```
T tab[n];
```

Identyfikator `tab` jest nazwą tablicy. W wyniku takiej deklaracji mamy dostępne w programie wartości `tab[0]`, `tab[1]`, ..., `tab[n-1]`, wszystkie tego samego typu `T`.

Wartość jaką przechowuje tablica `tab` jest wektorem wartości należącym do zbioru $T^n = T \times T \times \cdots \times T$. Wartość `tab[i]` jest i-tą składową tego wektora, dla $i \in \{0, 1, \ldots, n-1\}$.

W poniższym przykładzie zdefiniowano 10-cio elementową tablicę `perm`, w której umieszczono permutację $(10, 9, 8, \ldots, 1)$ liczb od 1 do 10:

```
int perm[10];
for(int i = 0; i < 10; i++)
  perm[i] = 10 - i;
```

Począwszy od roku 1999, istnieje w języku C możliwość definiowania *tablic o zmiennej długości.* C99

Tablice takie mają wielkość zadaną w chwili wykonywania się programu a nie podczas kompilacji. Nie oznacza to, że tablica taka może zmienić swoją wielkość. Po prostu zostaje ona utworzona podczas działania programu i dopóki istnieje w pamięci, nie zmienia swojego rozmiaru.

Na listingu 5.1 przedstawiono program, który dla danego całkowitego n, zlicza wszystkie liczby pierwsze z zakresu od 2 do n.

Listing 5.1: Sito Eratostenesa

```
// sito.c
//
// Program zlicza liczby pierwsze z przedziału [0 .. n]. Do
// szybkiego wyznaczania liczb pierwszych wykorzystano sito
// Eratostenesa.

#include <stdio.h>
#include <math.h>
#include <stdbool.h>
#include <assert.h>

int const max_n = 1000000;

int main(void)
{
  int n;
  printf("Podaj zakres (max %d): ", max_n);
  scanf("%d", &n);
  assert(n <= max_n);
  bool sito[n+1];
  for(int i = 0; i <= n; i++)
    sito[i] = true;
  sito[0] = false;
  sito[1] = false;
  int limit = sqrt (n);
  assert(limit <= n); // na potrzeby analizy statycznej

  for(int liczba = 0; liczba <= limit; liczba++)
  {
```

```
    if(sito[liczba])
    {
      for(int i = liczba*liczba; i <= n; i += liczba)
        sito[i] = false;
    }
  }

  int licznik = 0;
  for(int liczba = 0 ; liczba <= n; liczba++)
    if(sito[liczba])
      licznik++;

  printf("Znaleziono %d liczb pierwszych w [0 .. %d].\n",
         licznik, n);
  return 0;
}
```

W programie tym zastosowano sito Eratostenesa. Jest to algorytm, który w bardzo efektywny sposób znajduje wszystkie liczby pierwsze w zadanym zakresie.

Wykorzystuje on tablicę `bool sito[n+1]`, w której na i-tej pozycji zapisane jest czy liczba $i \in \{0, 1, \ldots, n\}$ może być liczbą pierwszą.

Początkowo wszystkie elementy tablicy `sito` przyjmują wartość **true**. Przed przystąpieniem do odsiewania liczb złożonych, na pozycjach 0 i 1 wpisywane są wartości **false**.

Podczas odsiewania, program szuka w pętli kolejnej wartości, która może być liczbą pierwszą (poszukuje w tablicy kolejnej wartości **true**). Jeśli znajdzie taką wartość i, to jest ona liczbą pierwszą (nie została wcześniej odsiana jako wielokrotność mniejszej liczby pierwszej). Zanim przystąpi się do poszukiwania kolejnej liczby pierwszej, należy odsiać wszystkie jej wielokrotności. Wykonuje to pętla, która począwszy od wartości $i \cdot i$ z krokiem i wpisuje w odpowiednie pozycje tablicy `sito` wartości **false**[1].

Na rysunku 5.1 przedstawiono pierwszych kilka iteracji powyższego algorytmu dla $n \geq 10$.

5.1.1 Arytmetyka wskaźników

Powinniśmy zdawać sobie sprawę, że tak naprawdę tablica w języku C, to tylko wskazanie na pierwszy jej element.

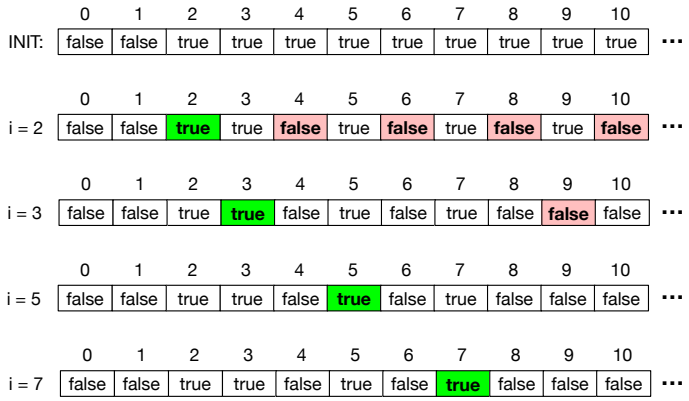
W poniższych dwóch linijkach programu:

```
int  tab1[3];
int* tab2;
```

obie zmienne są tego samego typu `int*` (wskazanie na liczbę całkowitą) a jedyną różnicą między nimi jest to, że przy deklaracji `tab1` zarezerwowanych zostanie 12 bajtów na przechowanie 3 liczb całkowitych.

Czym jest `tab1[2]` w przypadku powyższych deklaracji. Otóż kompilator widząc wyrażenie `tab1[2]` rozumie go jako `*(tab1 + 2)`. Operator wyłuskania (dereferencji) `*` już znamy. Czym zatem jest suma `tab1 + 2`. Otóż jest to wskazanie na miejsce pamięci, które jest o dwie długości wartości typu `int` dalej

[1] Zastanów się czemu od wartości i^2 a nie $2 \cdot i$.



Rysunek 5.1: Pierwsze iteracje algorytmu sita Eratostenesa.

niż miejsce wskazywane przez `tab1`. Jeśli więc `tab1` wskazuje na początek tablicy trzech liczb całkowitych, to `tab1 + 2` wskazuje na ostatnią liczbę w tablicy, zatem `*(tab1 + 2)` jest tą trzecią liczbą w tablicy.

Na rysunku 5.2 przedstawiono jak w pamięci przechowywane są elementy tablicy `tab1`.

To że kompilator języka C traktuje tablicę jako wskazanie niesie za sobą bardzo poważne konsekwencje. Podczas działania programu nie jest dokonywana kontrola zakresu indeksów tablicy. Zatem jeśli tablica `tab1` była zadeklarowana jako trzyelementowa, to program natrafiając na wyrażenie `tab1[4]` będzie próbował odczytać liczbę całkowitą leżącą **za** tablicą, natomiast natrafiając na wyrażenie `tab1[-2]` będzie próbował odczytać liczbę leżącą osiem bajtów **przed** tablicą. Powoduje to wiele problemów, gdyż prowadzi do trudnych do wykrycia błędów działania programu. Pół biedy jeśli odwołamy się do tak odległego elementu tablicy, że wykroczymy poza obszar pamięci przydzielony naszemu programowi. Wówczas program przerwie działanie z komunikatem błędu **Segmentation fault**. Gorzej jeśli miejsce do którego odwołamy się będzie leżeć w przydzielonym naszemu programowi obszarze pamięci. Wówczas możemy zniszczyć (nadpisać) inne dane. Często dowiemy się o tym otrzymując całkowicie bezsensowne wyniki albo gdy samolot uderzy w górę. A jeśli wyniki są sensowne? Czy możemy mieć do nich zaufanie, jeśli nie kontrolowano zakresu indeksów?

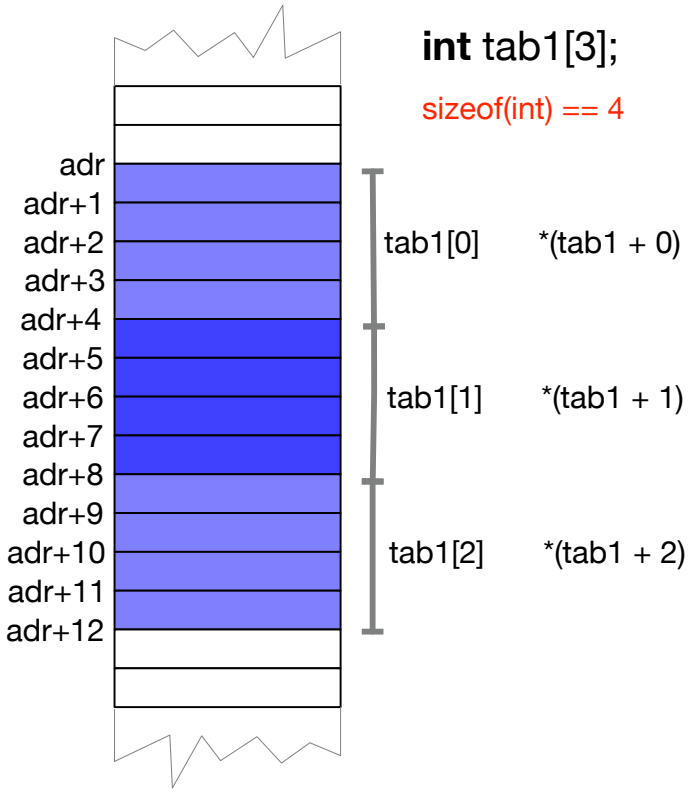
5.1.2 Łańcuchy znaków

Szczególnym zastosowaniem tablic jest przechowywanie łańcuchów znaków, tzn. napisów.

Jeśli chcemy przechowywać napis złożony z n znaków, to powinniśmy zadeklarować tablicę złożoną z $n + 1$ elementów typu `char`.

Dodatkowy element w tablicy potrzebny jest na przechowanie znaku o kodzie 0 oznaczającym koniec napisu (znak `'\0'`).

Każdy łańcuch znaków musi być w języku C zakończony znakiem `'\0'`.



Rysunek 5.2: Przechowywanie elementów tablicy w pamięci.

Wybrane funkcje operujące na łańcuchach znaków (wymagają dołączenia pliku nagłówkowego `string.h`):

`strlen(s)` długość napisu,

`strcmp(s1, s2)` porównanie alfabetyczne napisów,

`strcat(s1, s2)` scalenie dwóch napisów,

`strcpy(s1, s2)` skopiowanie napisu `s2` do `s1`.

W poniższym programie zastosowano tablicę znaków:

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

int main(void)
{
  char name[10];
  printf("What's your name? ");
  scanf("%s", name);
  printf("Hello %s! There are %lu letters in your name.\n",
      name, strlen(name));
  return 0;
}
```

Przykładowe uruchomienia:

```
$ ./main
What's your name? przemko
Hello przemko! There are 7 letters in your name.
$ ./main
What's your name? hermenegilda
Hello hermenegilda! There are 12 letters in your name.
Abort trap: 6
```


Komunikat **Abort trap: 6** oznacza, że wykryto pisanie w pamięci poza obszarem przeznaczonym na dane. Drugie imię miało długość większą niż 9 (tablica `name` jest 10-cio elementowa).

5.1.3 Tablice dwu- i więcej wymiarowe

W języku C dostępne są tablice wielowymiarowe. Deklaruje się je podając w nawiasach kwadratowych ich rozmiary w kolejnych wymiarach.

Tablicę dwuwymiarową o m wierszach i n kolumnach deklaruje się instrukcją:

```
T tab[m][n];
```

gdzie `T` jest typem wartości przechowywanych w tablicy.

Należy pamiętać, że zakres indeksów w każdym wymiarze zaczyna się od 0.

W poniższym fragmencie kodu zadeklarowano tabliczkę mnożenia i wypełniono są odpowiednimi wartościami:

```
int tabliczka[10][10];
for(int i = 0; i < 10; i++)
  for(int j = 0; j < 10; j++)
    tabliczka[i][j] = (i + 1) * (j + 1);
```

Zaleca się aby w odwołaniu `tab[i][j]` pierwszy indeks `i` traktować jako numer wiersza a drugi `j` jako numer kolumny.

W C można również deklarować tablice o większej liczbie wymiarów, np.

```
int kostka[5][5][5];    // trójwymiarowa kostka o 125 elementach
double parametr[5][6][7][8];      // tablica 1680-ciu parametrów
```

5.1.4 Inicjowanie elementów tablicy

W deklaracji tablicy można zainicjować wartości jej elementów. Dla tablic jednowymiarowych wystarczy wymienić wartości w nawiasach klamrowych oddzielając je przecinkami:

```
int cyfry_pierwsze[4] = {2, 3, 5, 7};
```

Jeśli inicjujemy wartości elementów tablicy, to można pominąć jej rozmiar a kompilator sam policzy ile ma ona elementów:

```
int cyfry_pierwsze[] = {2, 3, 5, 7};
```

W przypadku inicjowania tablic dwuwymiarowych, należy wartości podawać wierszami i grupować je w nawiasach klamrowych:

```
int kwadrat_magiczny[4][4] =
  { // równe sumy w każdym wierszu i każdej kolumnie
    {16,  3,  2, 13}, // 34
    { 5, 10, 11,  8}, // 34
    { 9,  6,  7, 12}, // 34
    { 4, 15, 14,  1}  // 34
     //////////////
//   34  34  34  34
  };
```


5.1.5 Problemy z pamięcią operacyjną

Pamiętajmy, że pamięć operacyjna komputera jest ograniczona. W systemie wielozadaniowym jakim jest np. Linux, musi ona być podzielona między wiele jednocześnie działających programów (procesów).

Każdy z procesów (w tym również twój program) dostaje od systemu pewien obszar pamięci na przechowywanie kodu programu oraz danych, na których ten kod operuje.

Przydzielony procesowi obszar danych podzielony jest na dwa ważne obszary:

STOS (ang. *stack*) przechowujący lokalne dane wywołanej funkcji aż do czasu jej zakończenia,

STERTA (ang. *heap*) przechowująca dane alokowane funkcją `malloc()`.

Koniecznie pamiętajmy, że obszar przeznaczony na stertę jest najczęściej dużo większym obszarem niż ten przeznaczony na stos (nawet o kilka rzędów wielkości).

Kiedy deklarujemy tablicę wewnątrz funkcji, to jest ona przechowywana na stosie. Ma to tę zaletę, że po zakończeniu działania funkcji zostanie ona usunięta ze stosu ale też ma wadę, bo zużywa cenne miejsce na niezbyt dużym stosie.

Dla przykładu, uruchamiając poniższy program dla $n = 10,000,000$:

```c
// tab1.c

#include <stdio.h>

int main(void)
{
  int n;
  scanf("%d", &n);
  int tab[n];
  for(int i = 0; i < n; i++)
    tab[i] = 13;
  return 0;
}
```

można zaobserwować dziwne jego działanie:

```
$ ./tab1
10000000
Segmentation fault: 11
```


świadczące o problemie z pamięcią potrzebą do przechowania tak dużej tablicy.

Sytuacja nie jest jednak beznadziejna. Pamiętając, że tablica to tak naprawdę tylko wskazanie na początek obszaru pamięci, w którym przechowywane są jej elementy, możemy skorzystać ze zmiennej wskaźnikowej:

```c
// tab2.c

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int n;
  scanf("%d", &n);
  int* tab = malloc(n * sizeof(int));
  if(tab == NULL)
  {
    printf("zabrakło pamięci!\n");
    return 1;
  }
  for(int i = 0; i < n; i++)
    tab[i] = 13;
  return 0;
}
```

Powyższy program przechowuje na stosie tylko dwie zmienne (`n` i `tab`), które na komputerze 64-bitowym zajmują 12 bajtów pamięci.

Cała tworzona przez wywołanie funkcji `malloc()` tablica przechowywana jest na stercie, a ta jest dużo większa od stosu:

```
$ ./tab2
10000000
$ ./tab2
100000000
$ ./tab2
1000000000
$ ./tab2
10000000000
$ ./tab2
100000000000
$ ./tab2
1000000000000
tab2(3713,0x119975dc0) malloc: can't allocate region
:*** mach_vm_map(size=18446744070800031744, flags: 100) failed (error code=3)
tab2(3713,0x119975dc0) malloc: *** set a breakpoint in malloc_error_break to debug
zabrakło pamięci!
```

Jak widać na stercie zmieściła się tablica stu miliardów liczb całkowitych i dopiero zabrakło miejsca dla biliona takich liczb (tysiąc miliardów).

5.2 Struktury

Wszystkie elementy tablicy są tego samego typu. Może się jednak zdarzyć, że potrzebujemy złożonej danej, której składniki są różnych typów. Możemy się w takim przypadku posłużyć strukturą (rekordem).

Deklaracja typu będącego strukturą złożoną z pól różnych typów ma w języku C postać:

```
struct S
{
  T1 p1;
  T2 p2;
  ...
  Tn pn;
};
```

gdzie `struct S` jest nazwą definiowanego typu a każde `Ti` i `pi` jest, odpowiednio, typem i nazwą i-tego pola struktury.

Po zadeklarowaniu typu `struct S` można deklarować zmienne tego typu:

```
struct S s1;
struct S s2, s3;
```

Zmienne takie przechowują dane złożone z wartości typów `T1`, `T2`, ..., `Tn`.

Wartość przechowywana w takiej strukturze jest wektorem należącym do zbioru:

$$T_1 \times T_2 \times \cdots \times T_n.$$

Rozpatrzmy następującą deklarację zmiennej `x`:

```
struct S x;
```

Aby dostać się do pola w strukturze przechowywanej w zmiennej `x` należy użyć selektora pola. W języku C ma on postać kropki, po której podaje się nazwę pola. Dla przykładu, `x.p1` jest pierwszym polem struktury `struct S` i przechowywana jest w nim wartość typu `T1`.

5.2.1 Inicjowanie struktury

Deklarując zmienną, której wartość jest strukturą, możemy jednocześnie zainicjować wartości pól tej struktury:

```
struct S s1 = {w1, w2, ..., wn};
```

gdzie `w1`, `w2`, ..., `wn` są wartościami kolejnych pól.

5.2.2 Przykład

Na listingu 5.2 przedstawiono program, w którym użyto struktur do reprezentowania punktów i kół na płaszczyźnie.

Listing 5.2: Reprezentacja punktu i koła.

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

struct punkt
{
  double x;
  double y;
};

struct kolo
{
  struct punkt srodek;
  double promien;
};

double odleglosc(struct punkt p1, struct punkt p2)
{
  return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2));
}

int main(void)
{
  struct punkt p1 = {0.0, 0.0};
  struct punkt p2 = {1.0, 1.0};
  struct kolo k1 = {p1, 0.5};
  struct kolo k2 = {p2, 0.9};

  if(k1.promien + k2.promien < odleglosc(k1.srodek, k2.srodek))
    printf("rozlaczne\n");
  else
    printf("maja czesc wspolna\n");
  return 0;
}
```

Poza odpowiednimi typami, zdefiniowano w nim również funkcję `odleglosc()`, która wylicza odległość euklidesową dwóch punktów na płaszczyźnie.

Znając odległość między środkami kół można sprawdzić czy są one rozłączne. Test ten ma postać następującego wyrażenia logicznego:

```
k1.promien + k2.promien < odleglosc(k1.srodek, k2.srodek)
```

gdzie `k1` i `k2` są dwiema zmiennymi przechowującymi struktury reprezentujące dwa koła na płaszczyźnie.

5.2.3 Rozmiar struktury

Pola struktury muszą często zaczynać się od adresów będących odpowiednią wielokrotnością rozmiaru ich typu. Tak jest np. w przypadku pól będących liczbami całkowitymi. Każda liczba całkowita musi być zapisana w pamięci pod adresem będącym wielokrotnością liczby 4. Powoduje to, że rozmiar struktury nie zawsze jest sumą rozmiarów jej pól.

Rozpatrzmy następujący program:

```c
#include <stdio.h>

int main(void)
{
  struct S {int i1; char c; int i2;};
  printf("sizeof(struct S) = %lu\n", sizeof(struct S));
  return 0;
}
```


Po jego uruchomieniu otrzymujemy wynik:

```
sizeof(struct S) = 12
```

Jak widać struktura ma rozmiar 12 bajtów, chociaż suma rozmiarów jej pól jest równa 9. Na rysunku 5.3 przedstawiono gdzie są brakujące trzy bajty pamięci.

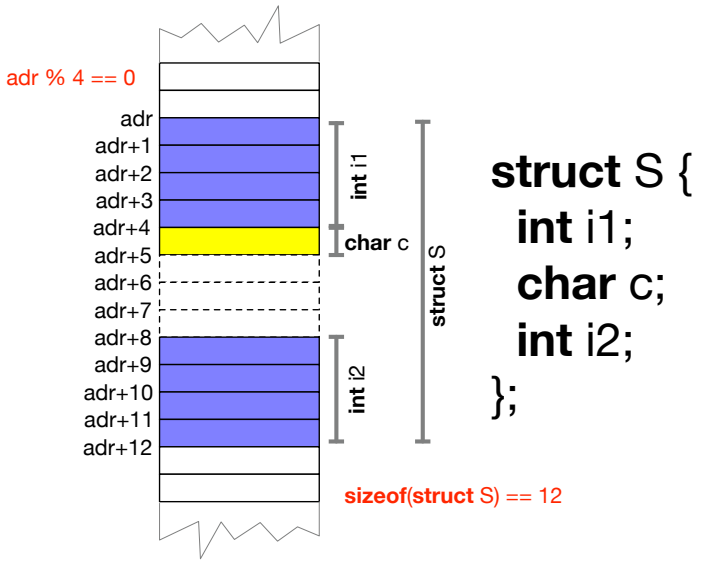
5.3 Unie

Czasami mamy do czynienia z sytuacją, w której zachodzi potrzeba reprezentowania danej, której typem jest suma mnogościowa skończonej liczby typów. Wówczas typ takiej danej możemy zapisać jako unię:

```c
union U
{
 T1 s1;
 T2 s2;
 ...
 Tn sn;
};
```

gdzie `union U` jest nazwą definiowanego typu a każde `Ti` i `si` jest i-tą składową tej unii o typie wartości `Ti` i nazwie składowej `si`.

Po zadeklarowaniu typu `union U` można deklarować zmienne tego typu:



Rysunek 5.3: Rozmiar struktury.


```
union U u1;
union U u2, u3;
```

Każda z tych zmiennych przechowuje dokładnie jedną wartość ze zbioru

$$T_1 \cup T_2 \cup \cdots \cup T_n.$$

Do składowych unii dostaje się za pomocą selektora w postaci kropki i nazwy składowej.

Rozmiarem danej typu `union U` jest największy spośród rozmiarów wartości typów `T1`, `T2`, ..., `Tn`.

5.3.1 Przykład

Rozpatrzmy następujące definicje struktur, unii i zmiennych:

```
struct Mężczyzna
{
  char imię[20];
  char nazwisko[20];
};

struct Kobieta
{
  char imię[20];
  char nazwisko[20];
  char nazwisko_panieńskie[20];
};

union Osoba
{
```

```
  struct Mężczyzna m;
  struct Kobieta k;
};

union Osoba pracownik;
```

Zmienną pracownik można użyć do przechowania danych o pracowniku, który osobą. Osoba ta jest albo mężczyzną albo kobietą. W zależności kim jest jej opis składa się z dwóch albo trzech pól.

Jeśli pracownik jest mężczyzną, to pole `pracownik.m.imię` przechowuje jego imię a pole `pracownik.m.nazwisko` przechowuje jego nazwisko.

Podobnie, jeśli pracownik jest kobietą, to pole `pracownik.k.imię` przechowuje jej imię, pole `pracownik.k.nazwisko` przechowuje jej nazwisko a pole `pracownik.k.nazwisko_panieńskie` jej nazwisko panieńskie.

Zauważ, że tak zdefiniowany typ do przechowywania osoby ma tę wadę, że nie wiemy czy przechowuje on dane o mężczyźnie czy o kobiecie.

Lepiej zdefiniować typ reprezentujący osobę następująco:

```
union Wariant
{
  struct Mężczyzna m;
  struct Kobieta k;
};

struct Osoba
{
  bool jest_mężczyzną;
  union Wariant w;
};
```


Tym razem typ reprezentujący osobę jest strukturą o dodatkowym polu boolowskim, w którym zapisano czy przechowywana w strukturze osoba jest mężczyzną czy nie.

Na listingu 5.3 przedstawiono przykładowy program korzystający z tak zdefiniowanych typów.

Listing 5.3: Warianty pól w strukturze.

```
// uni.c

#include <stdio.h>
#include <stdbool.h>
#include <string.h>

struct Mężczyzna
{
  char imię[20];
  char nazwisko[20];
};

struct Kobieta
{
  char imię[20];
  char nazwisko[20];
  char nazwisko_panieńskie[20];
```

```
};

union Wariant
{
  struct Mężczyzna m;
  struct Kobieta k;
};

struct Osoba
{
  bool jest_mężczyzną;
  union Wariant w;
};

void drukuj (struct Osoba o)
{
  if(o.jest_mężczyzną)
    printf("%s %s\n", o.w.m.imię, o.w.m.nazwisko);
  else
    printf("%s %s z domu %s\n", o.w.k.imię, o.w.k.nazwisko,
                                o.w.k.nazwisko_panieńskie);
}

int main(void)
{
  struct Osoba pracownik;
  pracownik.jest_mężczyzną = false;
  strcpy(pracownik.w.k.imię,             "Anna");
  strcpy(pracownik.w.k.nazwisko,         "Nowak");
  strcpy(pracownik.w.k.nazwisko_panieńskie, "Kowalska");
  drukuj(pracownik);
  return 0;
}
```


Przykład uruchomienia programu:

```
$ ./uni
Anna Nowak z domu Kowalska
```

5.4 Ciekawostka

O tym jak dziwnym językiem programowania jest język C świadczy następujący przykład.

Zastanówmy się czy poniższy program jest poprawny. Jeśli tak, to jaką liczbę wydrukuje?

```
#include <stdio.h>

int main(void)
{
  int tab[4] = {1, 3, 5, 7};
  printf("%d\n", 2[tab]);
  return 0;
}
```

Wątpliwość może budzić wyrażenie `2[tab]`. Pamiętając co mówiliśmy o arytmetyce wskaźników, możemy zinterpretować go następująco:

$$2[tab] = *(2 + tab) = *(tab + 2) = tab[2].$$

Wynika z tego, że program jest poprawny i wydrukuje trzecią liczbę z tablicy `tab`, która jest równa 5.