Podstawy programowania w języku C

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


double X(double t) {
    return (1.0 + sin(2.0 * M_PI * 10000.0 * t) + cos(2.0 * M_PI * 10000.0 * t));

    struct complexx {
        double Re;
        double Im;
    };

    struct complexx cmplx(double A, souble B) {
        struct complexx result;
        result.Re = A;
        result.Im = B;
        return (result);
    }


    struct complex cMult(struct complexx wA, struct complexx B) {

        struct complexx result;

        result.Re = A.Re * B.Im + A.Im * B.Re;
```

Struktury i unie

W języku C++, do definiowania własnych typów danych służą klasy. Jednak typy złożone istnieją również w czystym C. Mają one postać unii i struktur.

Lab9

Uniwersytet Rzeszowski
ur.edu.pl

Struktury

Struktura definiuje nowy typ danych. Jest to typ złożony: pojedyncza zmienna typu strukturalnego może zawierać wiele danych (liczb, napisów, wskaźników, itd.). W odróżnieniu od tablic, dane te nie muszą być tego samego typu (jak elementy tablicy).

C-struktura jest kolekcją nazwanych składowych, które mogą być różnych typów, również innych typów strukturalnych.

Definicja C-struktury może mieć następującą postać:

```
struct Nazwa {
        Typ1 sklad1;
        Typ2 sklad2;
        ...
};
```

Nazwy ***Typ1*** i ***Typ2*** są tu nazwami typów (innych niż nazwa typu strukturalnego który właśnie jest definiowany, czyli tutaj ***Nazwa***). Nazwy ***sklad1*** i ***sklad2*** dowolnymi identyfikatorami pól tej struktury. Każda zmienna tego typu strukturalnego będzie zawierać składowe o typach i nazwach odpowiadających polom struktury. Na razie jednak jest to tylko definicja typu: nie istnieją jeszcze żadne tego typu zmienne.

Średnik na końcu, po nawiasie zamykającym, jest konieczny!

W C++ po takiej samej definicji nazwą typu jest po prostu ***Nazwa***. Natomiast w C nazwą tego typu jest ***struct Nazwa*** — konieczne jest zatem powtórzenie słowa kluczowego ***struct*** przy późniejszym użyciu nazwy tego typu. Można tego uniknąć stosując standardową „sztuczkę" ze specyfikatorem ***typedef***:

```
typedef struct Nazwa {
    // definicja struktury
} Nazwa;
```

W ten sposób nazwa ***Nazwa*** staje się (już jednowyrazowym) aliasem pełnej nazwy ***struct Nazwa***.

Kiedy już zdefiniowaliśmy typ, możemy definiować zmienne tego typu; składnia jest dokładnie taka sama jak składnia deklaracji/definicji zmiennych typów wbudowanych (jak ***double*** czy ***int***):

```
struct Nazwa a, b;   /* C i C++ */
       Nazwa c, d;          // tylko C++
```

Można też definiować zmienne tego typu bezpośrednio za definicją, a przed średnikiem:

```
struct Nazwa {
        Typ1 skladowa1;
        Typ2 skladowa2;
} a, b, c, d;
```

definiuje typ strukturalny i od razu definiuje cztery zmienne tego typu. Ten ostatni zapis daje możliwość tworzenia obiektów struktur anonimowych:

```
struct {
        Typ1 skladowa1;
        Typ2 skladowa2;
} a, b;
```

Za pomocą takiej konstrukcji stworzyliśmy dwa obiekty strukturalne o nazwach ***a*** i ***b***. Każdy z nich zawiera dwie składowe odpowiadające dwóm polom w definicji struktury. Do składowych tych można się odnosić w sposób opisany poniżej, jak do składowych zwykłych struktur posiadających nazwę. Nie da się już jednak zdefiniować innych obiektów tego samego typu, bo typ ten nie ma żadnej nazwy i nie ma jak się do niego odwołać!

Każdy pojedynczy obiekt typu zdefiniowanego jako C-struktura zawiera tyle składowych i takich typów, jak to zostało określone w definicji tej C-struktury. Jeśli zdefiniowany (utworzony) jest obiekt, to do jego składowych odnosimy się za pomocą operatora wyboru składowej. Operator ten ma dwie formy (patrz pozycje 4 i 5 w tabeli operatorów), w zależności od tego, czy odnosimy się do obiektu poprzez jego nazwę, czy poprzez nazwę wskaźnika do niego. Jeśli ***a*** jest nazwą obiektu, to do jego składowej o nazwie ***sklad*** odnosimy się za pomocą operatora „kropki":

```
a.sklad
```

Ta sama reguła stosowałaby się, gdyby ***a*** było nazwą referencji do obiektu — pamiętajmy jednak, że w czystym C odnośników (referencji) nie ma. Natomiast jeśli ***pa*** jest wskaźnikiem do pewnego obiektu struktury, to do tej samej składowej odnosimy się za pomocą operatora „strzałki":

```
pa->sklad
```

(„strzałka" to dwuznak złożony z myślnika i znaku większości; odstęp między tymi dwoma znakami jest niedopuszczalny).

Ta sama zasada dotyczy nie tylko struktur, które mają postać C-struktur, ale wszystkich struktur i klas w C++.

Zauważmy, że forma ***pa->sklad*** jest w zasadzie notacyjnym skrótem zapisu ***(*pa).sklad***, gdyż ****pa*** jest właśnie l-wartością obiektu wskazywanego przez ***pa***. Zatem jeśli a jest identyfikatorem obiektu struktury posiadającej pole ***x*** typu ***double***, a ***pa*** wskaźnikiem do tego obiektu, to następujące instrukcje są równoważne:

```
a.x      = 3.14;
(&a)->x = 3.14;
```

```
pa->x   = 3.14;
(*pa).x = 3.14;
```
W lini 2 i 4 nawiasy są potrzebne, gdyż operatory wyboru składowej (kropka i „strzałka”) mają wyższy priorytet niż operatory dereferencji i wyłuskania adresu (' *' i ' &').

---

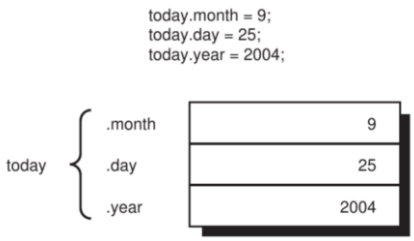
*Zapis **a.b** oznacza składową o nazwie **b** obiektu **a**. Zapis **pa->b** oznacza składową o nazwie **b** obiektu wskazywanego przez wskaźnik **pa**.*

Tworząc obiekt typu C-struktury można go od razu zainicjować. Składnia jest podobna do tej, jakiej używamy do inicjowania tablicy:

```
Nazwa ob = {wyr_1, wyr_2};
```

gdzie ***wyr_1*** i ***wyr_2*** są wyrażeniami, których wartości mają posłużyć do zainicjowania składowych obiektu w kolejności takiej, w jakiej zadeklarowane zostały odpowiednie pola struktury. Inicjatorów może być mniej niż pól; składowe odpowiadające pozostałym polom zostaną wtedy zainicjowane zerami odpowiedniego typu. Słowo kluczowe ***struct*** w powyższej instrukcji może być pominięte w C++, natomiast jest niezbędne w C.

Jak wspomniano wcześniej, struktura pozwala na przechowywanie elementów różnych typów w jednej zmiennej (możliwe jest również przechowywanie elementów tych samych typów w strukturze). Dostęp do tych elementów uzyskujemy za pomocą operatora kropki. Najprostszym przykładem na zrozumienie struktur, jest zdefiniowanie zmiennej, mającej możliwość przechowywania daty. Podając datę, zawsze musimy określić dzień, miesiąc oraz rok. Liczby odpowiadające konkretnemu dniu, miesiącowi czy rokowi przedstawić za pomocą liczb naturalnych, które w języku C mają swój odpowiednik w postaci typu danych ***unsigned int***. Ponieważ na datę składają się 3 liczby naturalne, nasza struktura będzie miała więc 3 pola: ***day***, ***month*** oraz ***year***. Dostęp do pól struktury (konieczny, gdy chcemy danemu polu przypisać wartość, lub po prostu wyświetlić dane pole) możliwy jest za pomocą operatora kropki, ***”.”*** Zamieszczony obrazek przedstawia graficznie definiowaną strukturę:



Na obrazku widzimy, iż za pomocą jednej zmiennej ***today*** mamy dostęp do wszystkich 3 pól struktury: ***day***, ***month*** oraz ***year***. Definicja struktury z obrazka przedstawiona została poniżej:

```
#include <stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    };

    return 0;
}
```

Wzorując się na zamieszczonym obrazku, utwórzmy teraz zmienną ***today***, oraz przypiszmy jej polom odpowiednie wartości. Poniższy kod przedstawia tworzenie zmiennej typu strukturalnego. Przed deklarację typu danych (tutaj jest to typ strukturalny, który nazwaliśmy ***date***), należy wstawić słowo kluczowe języka C, ***struct***, a dopiero później podać nazwę złożonego typu danych (***date***) oraz tak, jak przy deklaracji zmiennych, wskazać nazwę zmiennej:

```
#include <stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    };
    struct date today;

    return 0;
}
```

Kiedy zadeklarowaliśmy już zmienną ***today***, możemy każdemu jej polu przypisać wybrane wartości. (Gdybyśmy tego nie zrobili, w polach struktury znalazłyby się losowe informacje, obecne aktualnie w dostępnym obszarze pamięci. Aby to sprawdzić, odkomentuj linię printf("%d\n", today.month); oraz skompiluj i uruchom program).

```
#include<stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    };
    struct date today;

//printf("%d\n",today.month);
    today.day = 10;
    today.month = 1;
    today.year = 2022;

    printf("Wyswietlenie skladowych struktury:\n\n");

    printf("today.day=%d\n", today.day);
    printf("today.month=%d\n", today.month);
    printf("today.year=%d\n", today.year);
    return 0;
}
```

Tak samo, jak w przypadku typów wyliczeniowych, również i podczas deklarowania zmiennych strukturalnych możemy wykorzystać słowo kluczowe ***typedef*** aby skrócić zapis deklaracji zmiennej strukturalnej:

```
#include<stdio.h>

int main(int argc, char **argv) {
    typedef struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    } date;

    date today;

    today.day = 10;
    printf("today.day=%d\n", today.day);

    return 0;
}
```

Zmienną strukturalną możemy zadeklarować również bezpośrednio po deklaracji struktury, przez kończącym jej deklarację średnikiem. Zmienna zadeklarowana w ten sposób może być używana w programie w taki sam sposób, jak zmienna zadeklarowana z użyciem słowa kluczowego ***struct***, co pokazuje poniższy listing:

```
#include<stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    } today;
    today.day = 10;
    today.month = 1;
    today.year = 2022;
    printf("Wyswietlenie skladowych struktury:\n\n");
    printf("today.day=%d\n", today.day);
    printf("today.month=%d\n", today.month);
    printf("today.year=%d\n", today.year);
    return 0;
}
```

Poniższy listing zawiera zbiór przykładów reprezentujących możliwe deklaracje i definicje typów strukturalnych:

```
#include<stdio.h>

//1)DEKLARACJA STRUKTURY informujemy kompilator, ze utworzylismy typ daych w ktorym mozemy
przechowywac element typu int oraz element typu double
struct struktura_1 {
    int i;
    double d;
};
//2)DEKLARACJA STRUKTURY ORAZ DEFINICJA ZMIENNEJ s2 TYPU STRUKTURALNEGO ODRAZU PRZY DEKLARACJI
struct struktura_2 {
    char c;
    float f;
} s2;

//3)DEKLARACJA ANONIMOWEJ STRUKTURY - STRUKTURY NIEPOSIADAJACEJ NAZWY
struct {
    int i;
    char c;
    double tab[3]; //elementami struktury moga byc rowniez tablice
} sa1, sa2;//definicja zmiennych strukturalnych anonimowej struktury

//4)DEKLARACJA STRUKTURY z typedef
typedef struct struktura_3 {
    int i1;
    int i2;
};

//5)DEKLARACJA STRUKTURY z typedef
typedef struct struktura_4 {
    char c;
    double d;
    float f;
    int i;
} moja_struktura;
//moja_struktura bedzie nazwa nowego typu strukturalnego!


//6)DEKLARACJA ANONIMOWEJ STRUKTURY z typedef
```

```c
//nie do konca anonimowej, bo bedziemy sie do niej odnosic za pomoca nazwy 'struktura_5' - nazwy pomocniczej zastepczej
typedef struct {
    char c;
    double d;
    float f;
    int i;
} struktura_5;

struct przykladowaStruktura {
    char imie;
    char nazwisko;
    int wiek;
}osoba = {'K', 'M', 24};

struct DaneOsobowe {
    char imie[255];
    char nazwisko[255];
    int wiek;
};

int main(int argc, char **argv) {
//DEFINICJA STRUKTURY NR1 czyli rezerwacja pamieci (statyczna) dla struktury mozemy juz uzywac zmiennej s1 typu strukturalnego
    struct struktura_1 s1;
    s1.i = 5;
    s1.d = 3.14;
    printf("s1.i=%d, s1.d=%.2f\n", s1.i, s1.d);
//ze zmiennej s2 mozemy juz wczesniej korzystac, bo zostala utworzona wczesniej, czy deklaracji struktury 'struktura_2'
    s2.c = 'K';
    s2.f = 2.3;
    printf("s2.c=%c, s2.f=%.2f\n", s2.c, s2.f);
//korzystanie ze zmiennych anonimowej sturktury - tak samo jak przy zwyklej stukturze!
    sa2.c = 'K';
    sa2.i = 1;
    sa2.tab[0] = 3.14;
    sa2.tab[1] = 3.15;
    printf("sa2.tab[0]=%.2f, sa2.tab[1]=%.2f\n", sa2.tab[0], sa2.tab[1]);
//pomimo type def tu tez musimy uzyc slowa struct
    struct struktura_3 s3;
//piszemy juz bez slowa struct - dzieki typedef
    moja_struktura s4;
//piszemy juz bez slowa struct - dzieki typeef
    struktura_5 s5;
    printf("%c %c %d\n", osoba.imie, osoba.nazwisko, osoba.wiek);
//inicjalizacja struktury odrazu przy jej deklaracji
    struct DaneOsobowe os = {"Jan", "Kowalski", 36};
    printf("%s %s %d\n", os.imie, os.nazwisko, os.wiek);
    return 0;
}
```

Inicjalizacja struktury

Nic nie stoi na przeszkodzie, aby polom zmiennej złożonej (jeśli nie są wskaźnikami) przypisać określone wartości od razu przy ich definicji:

```c
#include<stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    } today = {10, 1, 2022};

    printf("Wyswietlenie skladowych struktury:\n\n");

    printf("today.day=%d\n", today.day);
    printf("today.month=%d\n", today.month);
    printf("today.year=%d\n", today.year);


    return 0;
}
```

```c
#include<stdio.h>

intmain(intargc,
char **argv
)
{
struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};
struct date today = {25, 9, 2004};
printf("Wyswietlenie skladowych struktury:\n\n");
printf("today.day=%d\n", today.day);
printf("today.month=%d\n", today.month);
printf("today.year=%d\n", today.year);
return 0;
```

```
}
```
Inicjalizując strukturę w powyższy sposób, należy pamiętać o kolejności pól w strukturze - kolejne liczby wymienione w nawiasach wąsatych zostaną przypisane odpowiadającym według kolejności polom struktury.
Możliwe jest również nadanie wartości elementom struktury poprzez przypisanie wartości dla każdego z jej pól oddzielnie. Poniższy kod zawiera przykłady inicjalizacji struktury, za pomocą obu omówionych sposobów:

```c
#include<stdio.h>

struct SimpleStruct {
    unsigned int ui;
    double d;
    float f;
};

int main(int argc, char **argv) {
//inicjalizacja struktury podczas jej definicji
    struct SimpleStruct s1 = {12, 3.14, 6.666};
    printf("ui=%u, d=%.2f, f=%.3f\n", s1.ui, s1.d, s1.f);

//inicjalizacja struktury podczas jej definicji
    struct SimpleStruct s2 = {.d=15.16, .ui=34};
    printf("ui=%u, d=%.2f, f=%.3f\n", s2.ui, s2.d, s2.f);

//inicjalizacja struktury tuz po jej definicji
    struct SimpleStruct s3;
    s3.ui = 1024;
    s3.d = 0.89;
    s3.f = 4.45;
    printf("ui=%u, d=%.2f, f=%.3f\n", s3.ui, s3.d, s3.f);

    return 0;
}
```

Struktury i funkcje

Struktury mogą być argumentami oraz typami zwracanymi funkcji. Do funkcji (struktura jako parametr) mogą być przekazywane przez wartość bądź przez wskaźnik, z funkcji (typ zwracany funkcji) mogą być zwracane również jako wartości lub wskaźniki. Do tej pory we wszystkich przykładach (oprócz przykładu wcześniejszego) wszystkie struktury deklarowaliśmy wewnątrz głównej funkcji programu, funkcji ***main***. Dzięki temu, zadeklarowany typ strukturalny był dostępny jedynie w obrębie tej funkcji, co udowadnia poniższy listing. Gdy spróbujemy zadeklarować zmienną typu ***date*** poza funkcją ***main***, kompilator poinformuje nas, iż taki typ nie jest mu znany:

```c
#include<stdio.h>

int main(int argc, char **argv) {
    struct date {
        unsigned int day;
        unsigned int month;
        unsigned int year;
    };

    struct date today = {25, 9, 2004};

    printf("Wyswietlenie skladowych struktury:\n\n");

    printf("today.day=%u\n", today.day);
    printf("today.month=%u\n", today.month);
    printf("today.year=%u\n", today.year);

    return 0;
}

struct date t;
int a;
```

```
C:/Users/Dawid/CLionProjects/untitled/main.c:22:13: error: storage size of 't' isn't known
   22 | struct date t;
      |             ^
ninja: build stopped: subcommand failed.
```

W tej sytuacji konieczne jest zdefiniowanie struktury poza jakąkolwiek funkcją w programie. Tak zadeklarowana struktura będzie widoczna w całym pliku *.c, dzięki czemu będzie też dostępna, jeśli zechcemy użyć jej jako argumentu bądź typu zwracanego funkcji:

```c
#include<stdio.h>

struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

int main(int argc, char **argv) {

    struct date today = {10, 1, 2022};

    printf("Wyswietlenie skladowych struktury:\n\n");

    printf("today.day=%u\n", today.day);
    printf("today.month=%u\n", today.month);
    printf("today.year=%u\n", today.year);

    return 0;
}

struct date t;
int a;
```

Jeśli w programie zadeklarujemy strukturę:

```c
struct date {
    unsigned int day;
```

```
    unsigned int month;
    unsigned int year;
};
```
Możemy użyć jej jako argumentu funkcji:
```
void funkcja1(struct date today){
//...
}
```
Albo typu zwracanego funkcji:
```
struct date funkcja2(int i, double d) {
//...
}
```
Oczywiście możemy również zadeklarować funkcję, która przyjmuje jako argument strukturę, oraz zwraca strukturę:
```
struct date funkcja3(struct date today) {
//...
}
```
Poniższy kod jest przykładem wykorzystania typu złożonego jako argumentu funkcji. Do funkcji przekazujemy przez wartość zmienną strukturalną, zadeklarowaną wcześniej w funkcji ***main***, a następnie funkcja zajmuje się wyświetleniem zawartości jej pól.
```
#include<stdio.h>

struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

void ShowDate(struct date today) {
    printf("Wyswietlenie skladowych struktury:\n\n");

    printf("today.day=%u\n", today.day);
    printf("today.month=%u\n", today.month);
    printf("today.year=%u\n", today.year);
}

int main(int argc, char **argv) {
    struct date today = {10, 1, 2022};
    ShowDate(today);
    return 0;
}
```
Identyczny do powyższego przykład, którego jedyną różnicą jest przekazywanie argumentu (struktury) przez wskaźnik, zamiast przekazywania przez wartość, znajduje się poniżej. Należy zwrócić uwagę, iż jeśli struktura przekazana jest do funkcji za pomocą wskaźnika, dostęp do elementów struktury możliwy jest jedynie za pomocą operatora **„->”**, a nie za pomocą operatora **„.”**:
```
#include<stdio.h>

struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

void ShowDate(struct date *today) {
    printf("Wyswietlenie skladowych struktury:\n\n");
    printf("today->day=%u\n", today->day);
    printf("today->month=%u\n", today->month);
    printf("today->year=%u\n", today->year);
}

int main(int argc, char **argv) {
    struct date today = {10, 1, 2022};
    ShowDate(&today);
    return 0;
}
```
W kolejnym przykładzie pokażemy wykorzystanie struktury jako typu zwracanego funkcji. W tym celu zadeklarujemy strukturę będącą w stanie przechowywać aktualną godzinę w formacie: HH:MM:SS. Nasza funkcja będzie pobierać jako argument aktualny czas, a następnie będzie zwiększać ilość sekund, oraz zwracać nowo utworzoną strukturę, zmodyfikowaną o jedną sekundę. Oczywiście należy uwzględnić przypadek, gdy ilość sekund będzie wynosić 59, wówczas należy zmodyfikować również ilość minut. Analogiczny warunek musi być uwzględniony podczas zwiększania minut czy godzin.
```
#include<stdio.h>

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

struct time UpdateTime(struct time now) {
```

```
    ++now.seconds;
//lub inaczej:
//now.seconds+=1;
//albo:
//now.seconds=now.seconds+1;
    if (now.seconds == 60) {
        now.seconds = 0;
        ++now.minutes;
//kolejna minuta
        if (now.minutes == 60) {
//kolejna godzina
            now.minutes = 0;
            ++now.hour;
        }
    }
//polnoc
    if (now.hour == 24)
        now.hour = 0;
    return now;
}

int main(int argc, char **argv) {
    struct time now = {.hour=23, .minutes=41, .seconds=59};
    printf("%u:%u:%u\n", now.hour, now.minutes, now.seconds);
    now = UpdateTime(now);
    printf("%u:%u:%u\n", now.hour, now.minutes, now.seconds);
    return 0;
}
```

Przeanalizujmy powyższy kod. W liniach 3-8 deklarujemy nowy, złożony typ danych, ***time***. Następnie podajemy deklarację i definicję funkcji - prostego zegara, który zwiększa aktualny czas o jedną sekundę. Funkcja przyjmuje jako argument czas, aktualizuje go, a następnie zwraca tak zmodyfikowaną zmienną. Następnie, w głównej funkcji naszego programu, funkcji ***main*** tworzymy zmienną ***now***, której to polom przypisujemy odpowiednie wartości, aby je kolejno wyświetlić. W linii 44 wywołujemy funkcję aktualizującą czas, podając jej jako argument wcześniej utworzoną zmienną strukturalną ***now***, natomiast wynik zwrócony przez funkcję przypisujemy również do zmiennej ***now***. W naszym przypadku możemy to zrobić, jest to jak najbardziej bezpieczna operacja, ponieważ składowymi struktury są zwykłe zmienne, a nie wskaźniki. Kiedy więc wykonamy przypisanie zmiennej strukturalnej do innej zmiennej strukturalnej, kompilator automatycznie przepisze wartości odpowiednich pól, co ilustruje poniższy, prosty przykład:

```
#include<stdio.h>

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

int main(int argc, char **argv) {
    struct time t1, t2;
    t1.hour = 23;
    t1.minutes = 53;
    t1.seconds = 24;
//zostana wyswietlone wartosi przypisane odp. polom
    printf("t1=%u:%u:%u\n", t1.hour, t1.minutes, t1.seconds);
//zostana wyswietlone losowe liczby, bo t2 jest niezainicjalizowane
    printf("t2=%u:%u:%u\n", t2.hour, t2.minutes, t2.seconds);
//odpowiadajace pole z t1 zostanie przypisane polu t2
//dziala to tak:
//t2.hour=t1.hour
//t2.minutes=t1.minutes
//t2.seconds=t1.seconds
    t2 = t1;
//zostana wyswietlone wartosci przypisane odp. polom
    printf("t1=%u:%u:%u\n", t1.hour, t1.minutes, t1.seconds);
//zostana wyswietlone wartosci przypisane odp. polom
    printf("t2=%u:%u:%u\n", t2.hour, t2.minutes, t2.seconds);
    return 0;
}
```

Tablice struktur

Jak pokazano w powyższych przykładach, struktury są typem danych mogącym w jednej zmiennej przechowywać jednocześnie kilka zmiennych różnych (choć niekoniecznie) typów danych. Jeśli chcielibyśmy w programie przechowywać więcej niż jedną datę (złożony typ danych), możemy do tego celu wykorzystać zmienne typu strukturalnego:

```
#include <stdio.h>

struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

int main(int argc, char **argv) {
    struct date d1 = {13, 1, 2022};
```

```
    struct date d2 = {14, 1, 2022};
    struct date d3 = {15, 1, 2022};

    return 0;
}
```

Jeśli zechcielibyśmy w programie przechowywać 1000 dat, wówczas konieczne byłoby zadeklarowanie 1000 zmiennych typu strukturalnego. Takie rozwiązanie nie jest dobrym pomysłem - wymaga mnóstwo pisania, a co za tym idzie, powtarzalności dużej ilości kodu, co jest zwyczajnie niepraktyczne. Zamiast tego, możemy wykorzystać inną znaną z języka C konstrukcję - tablicę. Przechowywanie struktur w tablicy znacznie uprości zapis oraz operowanie wieloma datami (zmiennymi typu strukturalnego) w programie. Deklarując tablicę struktur utworzymy tablicę, której każdym elementem będzie zmienna typu strukturalnego. Możemy tworzyć dynamiczne i statyczne tablice struktur.

```
#include <stdio.h>

struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

int main(int argc, char **argv) {
//najpierw definicja
    struct date tab1[3];

// pozniej deklaracja, przypisanie wartosci
elementom tablicy-strukturom
    tab1[0].day = 13;
    tab1[0].month = 1;
    tab1[0].year = 2022;

    tab1[1].day = 14;
    tab1[1].month = 1;
    tab1[1].year = 2022;

    tab1[2].day = 15;
    tab1[2].month = 1;
    tab1[2].year = 2022;

// lub, inaczej, deklaracja odrazu przy
definicji:
    struct date tab2[3] = {{13, 1, 2022},
                           {14, 1, 2022},
                           {15, 1, 2022}};

    return 0;
}
```

Deklaracja statycznej tablicy struktur wygląda następująco:

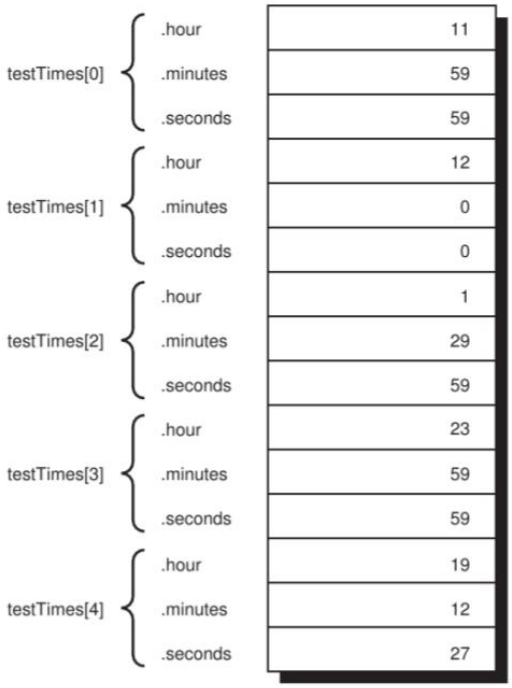
```
#include <stdio.h>

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

int main(int argc, char **argv) {
    struct time testTimes[5];

    return 0;
}
```

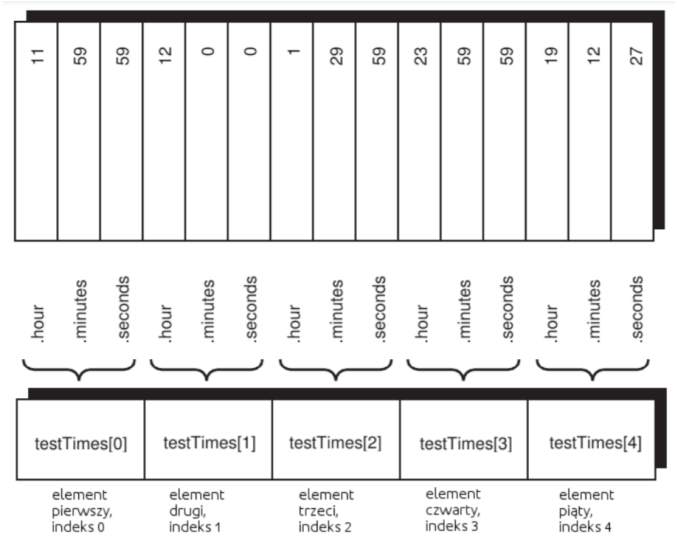
Przeanalizujmy powyższy kod. Tworząc tablicę struktur należy poprzedzić ją słowem kluczowym ***struct*** (chyba, że strukturę zadeklarowaliśmy ze słowem kluczowym ***typedef***, wówczas nie jest to konieczne), podać nazwę typu złożonego (tutaj ***time***) oraz nazwę zmiennej tablicowej, przechowującej struktury (u nas jest to ***testTimes***) a także ilość przechowywanych w tablicy struktur (w przykładzie mamy 5 struktur w tablicy, ***testTimes[5]***).

Do kolejnych komórek tablicy (0,1,2,3,4), które są strukturami, możemy wpisywać wybrane wartości. Aby to zrobić używamy nazwy zmiennej (tablicy), podajemy indeks (numer struktury w tablicy, do której chcemy wpisać wartości), oraz kolejne pola, do których wpiszemy dane):

```
#include <stdio.h>

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

int main(int argc, char **argv) {
    struct time testTimes[5];

    testTimes[0].hour = 11;
    testTimes[0].minutes = 59;
    testTimes[0].seconds = 59;

//…

    testTimes[3].hour = 23;
    testTimes[3].minutes = 59;
    testTimes[3].seconds = 59;

    printf("%u:%u:%u\n",
           testTimes[0].hour,
           testTimes[0].minutes,
           testTimes[0].seconds);

    return 0;
}
```

W pamięci tablicę struktur można przedstawić następująco:





Wiemy już, jak utworzyć statyczną, jednowymiarową tablicę struktur. Spróbujmy zatem stworzyć dynamiczną tablicę struktur, przydzielić na nią pamięć, wypełnić wartościami, a następnie usunąć przydzieloną pamięć. Dynamiczna tablica struktur będzie miała identyczny rozmiar i elementy, jak statyczna tablica struktur. Jedyną różnicą będzie jej „dynamiczność".
Deklaracja dynamicznej tablicy struktur wygląda następująco:

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

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

int main(int argc, char **argv) {
    struct time *testTimes;
    testTimes = malloc(sizeof(struct time) * 5);
    free(testTimes);
    return 0;
}
```

Do kolejnych komórek tablicy (0,1,2,3,4), które są strukturami, możemy wpisywać wybrane wartości. Aby to zrobić używamy nazwy zmiennej (tablicy), podajemy indeks (numer struktury w tablicy, do której chcemy wpisać wartości), oraz kolejne pola, do których wpiszemy dane):

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

struct time {
    unsigned int hour;
    unsigned int minutes;
    unsigned int seconds;
};

int main(int argc, char **argv) {
    struct time *testTimes;
    testTimes = malloc(sizeof(struct time) * 5);
    testTimes[0].hour = 11;
    testTimes[0].minutes = 59;
    testTimes[0].seconds = 59;
//…
    testTimes[3].hour = 23;
    testTimes[3].minutes = 59;
    testTimes[3].seconds = 59;

    printf("%u:%u:%u\n", testTimes[0].hour, testTimes[0].minutes, testTimes[0].seconds);
    free(testTimes);

    return 0;
}
```

Funkcje korzystające z tablic struktur

Funkcje mogą korzystać z dynamicznych i statycznych tablic struktur. Dynamiczna lub statyczna tablica struktur może być zarówno argumentem jak i typem zwracanym funkcji.

Funkcja korzystająca ze statycznej tablicy struktur: Poniższy przykład pokazuje, iż wykorzystanie statycznej tablicy struktur w funkcji nie różni się znacząco od wykorzystania „zwykłej" statycznej tablicy w funkcji. Jedyną różnicą jest konieczność wykorzystania słowa kluczowego ***struct*** oraz konieczność użycia operatora kropki podczas dostępu do pola struktury (elementu tablicy).

```
#include <stdio.h>

struct ksiazka {
    double cena;
    char tytul[255];
};

double SumaCen(struct ksiazka tab[], unsigned int n) {
    unsigned int i;
    double suma = 0.0;

    for (i = 0; i < n; i++)
        suma += tab[i].cena;

    return suma;
}

int main(int argc, char **argv) {

    struct ksiazka tab[2] = {{120, "Programowanie w C"},
                             {150, "Programowanie w C++"}};

    printf("%.2f PLN\n", SumaCen(tab, 2));

    return 0;
}
```

Funkcja korzystająca z dynamicznej tablicy struktur:

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

struct ksiazka {
    double cena;
    char tytul[255];
};

double SumaCen(struct ksiazka *tab, unsigned int n) {
    unsigned int i;
    double suma = 0.0;

    for (i = 0; i < n; i++)
        suma += tab[i].cena;

    return suma;
}

int main(int argc, char **argv) {

    struct ksiazka *tab = malloc(sizeof(struct ksiazka) * 2);

    tab[0].cena = 120;
    strcpy(tab[0].tytul, "Programowanie w C");
    tab[1].cena = 150;
    strcpy(tab[1].tytul, "Programowanie w C++");

    printf("%.2f PLN\n", SumaCen(tab, 2));

    free(tab);

    return 0;
}
```

Struktury zawierające tablice

Ponieważ polami struktury mogą być zmienne dowolnego typu danych istniejącego w języku C, nic nie stoi na przeszkodzie, aby polem struktury była tablica. Utwórzmy strukturę, która przechowywać będzie nazwę miesiąca - właściwie, skrót tej nazwy (w postaci tablicy znaków), oraz ilość dni danego miesiąca (liczba naturalna). W pierwszym przykładzie wykorzystamy tablicę statyczną, jako składnik struktury, w drugim zaś tablicę dynamiczną (w strukturze znajdzie się wskaźnik).

W pierwszym przykładzie wykorzystujemy statyczną tablicę znaków jako pole struktury:

```
#include <stdio.h>

struct month {
    unsigned int numberOfDays;
    char name[3];
};

int main(int argc, char **argv) {
// deklaracja
    struct month aMonth;

// definicja
    aMonth.numberOfDays = 31;
    aMonth.name[0] = 'J';
    aMonth.name[1] = 'a';
    aMonth.name[2] = 'n';

// definicja i deklaracja jednoczesnie
    struct month bMonth = {21, {'F', 'e',
'b'}};

    printf("%s has %u days. \n", aMonth.name,
aMonth.numberOfDays);
    printf("%s has %u days. \n", bMonth.name,
bMonth.numberOfDays);

    return 0;
    }
```

W drugim przykładzie wykorzystujemy dynamiczną tablicę znaków jako pole struktury:

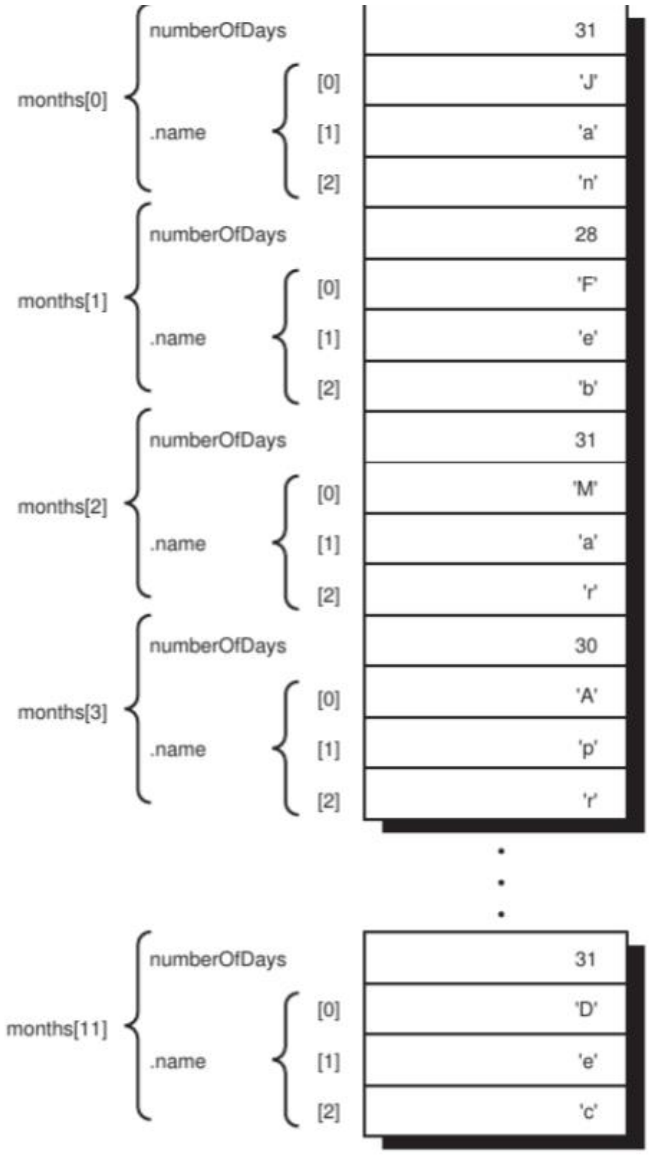
```
#include <stdio.h>
#include <malloc.h>

struct month {
    unsigned int numberOfDays;
    char *name;
};

int main(int argc, char **argv) {
// deklaracja
    struct month aMonth;

// definicja
    aMonth.numberOfDays = 31;

// przydziel pamiec na tablice dynamiczna
    aMonth.name = malloc(sizeof(char) * 3);

// definicja
    aMonth.name[0] = 'J';
    aMonth.name[1] = 'a';
    aMonth.name[2] = 'n';

// deklaracja
    struct month bMonth;

// definicja
    bMonth.numberOfDays = 21;

// przydziel pamiec na tablice dynamiczna
    bMonth.name = malloc(sizeof(char) * 3);

// definicja
    bMonth.name[0] = 'F';
    bMonth.name[1] = 'e';
    bMonth.name[2] = 'b';

    printf("%s has %u days. \n", aMonth.name,
aMonth.numberOfDays);
    printf("%s has %u days. \n", bMonth.name,
bMonth.numberOfDays);

// zwolnij pamiec
    free(aMonth.name);
    free(bMonth.name);

    return 0;
}
```

Możemy również zadeklarować tablicę struktur, której polem jest inna tablica. Deklaracja takiej tablicy struktur nie różni się niczym od deklaracji tablicy struktur, które nie zawierają tablicy jako jednego ze swoich pól. Korzystając z powyższych przykładów, możemy zadeklarować tablicę struktur, która będzie przechowywała informacje o 12 miesiącach roku - skróty ich nazw oraz liczbę dni każdego miesiąca. Nasza tablica struktur będzie miała 12 elementów, każdym z jej elementów będzie struktura, która jako pole będzie posiadała tablicę znaków.

```
#include <stdio.h>

struct month {
    unsigned int numberOfDays;
    char name[3];
};

int main(int argc, char **argv) {
    int i;

// deklaracja i definicja
    struct month months[12] = {{31, 'J', 'a',
'n'},
                               {21, 'F', 'e',
'b'},
                                {31, 'M', 'a',
'r'}};

// definicja
            months[3].numberOfDays = 30;
            months[3].name[0] ='A';
            months[3].name[1] ='p';
            months[3].name[2] ='r';

// wyswietlenie tablicy struktur
            for (i = 0; i < 3; ++i)
            printf(" %c%c%c \t%u\n",
months[i].name[0], months[i].name[1],
months[i].name[2], months[i].numberOfDays);

            return 0;
    }
```

Tablicę struktur możemy zdefiniować od razu przy jej definicji (linia 14), podając w kolejnych nawiasach dane, które mają znaleźć się w odpowiednich komórkach tablicy. Nawias główny zawiera wówczas wewnątrz podnawiasy, które zostaną użyte do zainicjalizowania kolejnych elementów tablicy, które są strukturami. Przykładowo, komórka tablicy o indeksie 0, to nasza pierwsza z 12 struktur. Zostanie ona zainicjalizowana wartościami 31 oraz' "Jan' (od angielskiego January). Możemy również (linia 20)

zainicjalizować komórkę tablicy (strukturę) po deklaracji. Pierwszy użyty indeks, tu: ***months[3]*** wskazuje, iż będziemy uzupełniać 3 komórkę tablicy struktur, zaś drugi w kolejności indeks (***months[3].name[1]***) odnosi się do tablicy, która jest polem struktury. Tak zdefiniowaną tablicę struktur w pamięci można przedstawić następująco:



Statyczne i dynamiczne zmienne strukturalne

Poniższy kod zawiera przykładową deklarację, definicję i wykorzystanie zmiennych strukturalnych zarówno statycznych, jak i dynamicznych (dostępnych za pomocą wskaźnika).

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

typedef struct trojkat {
    int a, b, c;
} trojkat;

int main(int argc, char **argv) {
// przydzial pamieci dla dynamicznej zmiennej strukturalnej
    struct trojkat *t1 = malloc(sizeof(trojkat));

// przypisanie wartosci polom dynamicznej zmiennej strukturalnej
    t1->a = 3;
    t1->b = 6;
    t1->c = 7;

// wyswietlenie (dostepdo) wartosci pol dynamicznej zmiennej strukturalnej
    printf(" t1->a = %d, t1->b = %d, t1->c = %d\n", t1->a, t1->b, t1->c);

// zwolnienie pamieci przydzielonej dynamicznej zmiennej struktralnej, przydzielonej w linii 10
    free(t1);

// przydzial pamieci dla statycznej zmiennej strukturalnej - zajmie sie tym kompilator
    struct trojkat t2;

// przypisanie wartosci polom statycznej zmiennej strukturalnej
    t2.a = 35;
    t2.b = 63;
    t2.c = 71;

// wyswietlenie (dostep do) wartosci pol statycznej zmiennej strukturalnej
    printf(" t2.a = %d, t2.b = %d, t2.c = %d\n", t2.a, t2.b, t2.c);

// zwolnieniem pamieci przydzielonej statycznej zmiennej strukturalnej zajmie sie kompilator, nie musimy nic pisac
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>


struct przykladowa {
    char znak;
    int calkowita;
};

int main() {
    int i, suma = 0;

// zmienna typu strukturalnego
    struct przykladowa p;
// dostep do elementow
    p.calkowita = 1024;
    p.znak = 'R';
    printf("%d %c \n", p.calkowita, p.znak);

/*
********************************************
****** */

// wskaznik na zmienna typu strukturalnego
    struct przykladowa *wp;
// wskazuje na p
    wp = &p;
// uzywamy -> aby dostac sie do elementu
    printf("%d %c \n", wp->calkowita, wp->znak);

/*
********************************************
****** */

// tablica struktur - statyczna, jednowymiarowa
    struct przykladowa tab[2];
// dostep do elementow
    tab[0].calkowita = 100;
    tab[0].znak = 'K';
    tab[1].calkowita = 2;
    tab[1].znak = 'M';
    printf("%d %c \n", tab[0].calkowita, tab[0].znak);
    for (i = 0; i < 2; i++)
        printf("%d %c ", tab[i].calkowita, tab[i].znak);

/*
********************************************
****** */

// tablica wskaznikow do struktur
    struct przykladowa *tabw[2];
// dostep do elementow
    tabw[0] = malloc(sizeof(struct przykladowa) * 3);
    for (i = 0; i < 3; i++)
        tabw[0][i].calkowita = 100;
    for (i = 0; i < 3; i++)
        suma += tabw[0][i].calkowita;
    printf("\nsuma = %d\n", suma);
    free(tabw[0]);

    return 0;
}
```

Podsumowanie - struktury

Poniższe kody zawierają przykłady definicji i użycia:

- zwykłej zmiennej strukturalnej
- wskaźnika do zmiennej strukturalnej
- statycznej tablicy struktur
- dynamicznej tablicy struktur
- tablicy wskaźników na struktury (inaczej: statyczna tablica wskaźników na dynamiczne struktury)

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


typedef struct Student {
    char imie[255];
    char nazwisko[255];

} Student;

int main() {

/* zmienna s typu Student */
    struct Student s;
    strcpy(s.imie, " Aleksander");
    strcpy(s.nazwisko, " Kowalski ");
    printf("Ze zmiennej : \t\t\t\t\t\t%s %s \n", s.imie, s.nazwisko);

/* poj. wskaznik na zmienna strukturalna Student */
    struct Student *w;
/* wskazuje na zmienna s typu Student */
    w = &s;
    printf("Ze wskaznika : \t\t\t\t\t\t%s %s \n", w->imie, w->nazwisko);

/* statyczna tablica 10 zmiennych typu Student */
    struct Student tab[10];
    tab[0] = s;
    printf("Ze statycznej tablicy : \t\t\t\t%s %s \n", tab[0].imie, tab[0].nazwisko);

/* dynamiczna tablica 10 zmiennych typu Student */
    struct Student *wsk = malloc(sizeof(struct Student) * 10);
```

```
        wsk[0] = s;
        printf("Z dynamicznej tablicy : \t\t\t\t\t%s %s \n", wsk[0].imie, wsk[0].nazwisko);
        free(wsk);

    /* tablica 20 wskanikow na zmienne typu Student */
        struct Student *tabwsk[20];
        tabwsk[0] = &s;
        tabwsk[1] = malloc(sizeof(struct Student) * 1);
        strcpy(tabwsk[1]->imie, "Adam");
        strcpy(tabwsk[1]->nazwisko, "Nowak");
        printf("Z tablicy wskaznikow na struktury: \t\t\t%s %s \n", tabwsk[0]->imie, tabwsk[0]->
    nazwisko );
        printf("Z tablicy wskaznikow na struktury: \t\t\t%s %s \n", tabwsk[1]->imie, tabwsk[1]->
    nazwisko );
        free(tabwsk[1]);

        return 0;
    }

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

typedef struct Student {
    char imie[255];
    char nazwisko[255];

} Student;

void fun1(struct Student s) {
    printf("Ze zmiennej: \t\t\t\t\t%s %s \n", s.imie, s.nazwisko);
}

void fun2(struct Student *wsk) {
    printf("Ze zmiennej: \t\t\t\t\t%s %s \n", wsk->imie, wsk->nazwisko);
}

void fun3(struct Student tab[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("Ze statycznej tablicy: \t\t\t\t%s %s \n", tab[i].imie, tab[i].nazwisko);
}

void fun4(struct Student *wsk, int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("Z dynamicznej tablicy: \t\t\t\t%s %s \n", wsk[i].imie, wsk[i].nazwisko);
}

void fun5(struct Student *tabwsk[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("Z tablicy wskaznikow na struktury: \t\t\t%s %s \n", tabwsk[i]->imie, tabwsk[i]-
>nazwisko);
}

int main() {

/* zmienna s typu Student */
    struct Student s;
    strcpy(s.imie, "Aleksander");
    strcpy(s.nazwisko, "Kowalski");
    fun1(s);

/* poj. wskaznik na zmienna strukturalna Student */
    struct Student *w;
/* wskazuje na zmienna s typu Student */
    w = &s;
    fun2(w);

/* statyczna tablica 10 zmiennych typu Student */
    struct Student tab[10];
    tab[0] = s;

/* dynamiczna tablica 10 zmiennych typu Student */
    struct Student *wsk = malloc(sizeof(struct Student) * 10);
    wsk[0] = s;
```

```
    free(wsk);

/* tablica 2 wskanikow na zmienne typu Student */
    struct Student *tabwsk[2];
    tabwsk[0] = &s;
    tabwsk[1] = malloc(sizeof(struct Student) * 1);
    strcpy(tabwsk[1]->imie, "Adam");
    strcpy(tabwsk[1]->nazwisko, "Nowak");
    fun5(tabwsk, 2);
    free(tabwsk[1]);

    return 0;
}
```

Unie

Unia nie jest właściwie strukturą danych, chociaż definiujemy ją podobnie:

```
union nazwa_typu_unii
{
  typ pole_1;
  typ pole_2;
  ...
  typ pole_n;
};
```

Po zdefiniowaniu typu unii, wykorzystujemy go do definicji zmiennych:

```
union nazwa_typu_unii zmienna;
```

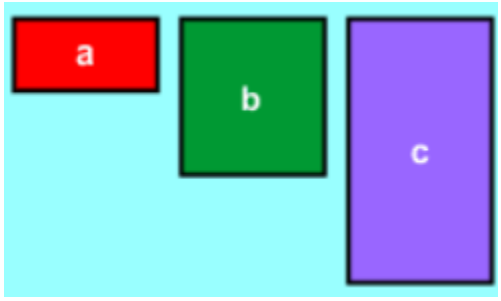
Zmienne można również definiować bez nazywania typu unii:

```
union  {
  typ pole_1;
  typ pole_2;
  ...
  typ pole_n;
} zmienna;
```

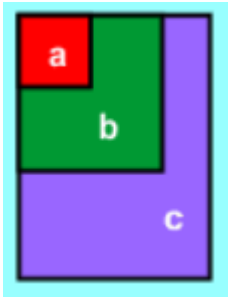
W takim przypadku powstaje zmienna unii posiadająca podane w definicji pola.

Struktura pozwala na jednoczesne przechowywanie wielu danych w swoich polach, ponieważ każde z nich w pamięci jest umieszczone w innym miejscu. Unia pozwala na przechowywanie tylko jednej danej naraz, ponieważ wszystkie pola unii są umieszczone w tym samym miejscu w pamięci.

```
struct xx {
  char a;
  int b;
  float c;
};
```




```
union yy {
  char a;
  int b;
  float c;
};
```



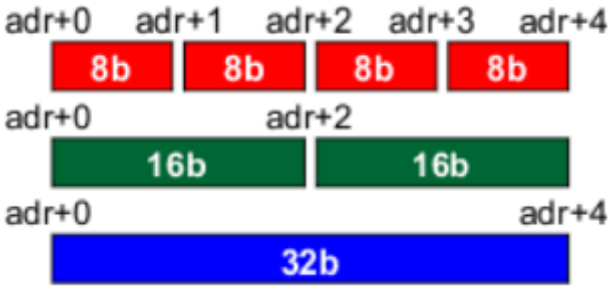
W strukturze ***xx*** mamy trzy pola ***a***, ***b*** i ***c***. W tych polach program może umieścić trzy różne dane i struktura będzie je przechowywała. W unii ***yy*** również mamy trzy pola: ***a***, ***b*** i ***c***. Jednak zajmują one ten sam obszar pamięci, dlatego w danej chwili unia może przechowywać dane tylko w jednym z tych pól. Unia pozwala efektywnie wykorzystać pamięć, szczególnie wtedy, gdy jest jej mało. Dzięki niej istnieje możliwość wykorzystania tego samego obszaru pamięci do różnych celów.

W pamięci struktura zajmuje taki obszar, aby pomieścił wszystkie jej pola. Unia natomiast zajmuje taki obszar, aby pomieścił jej największe pole.

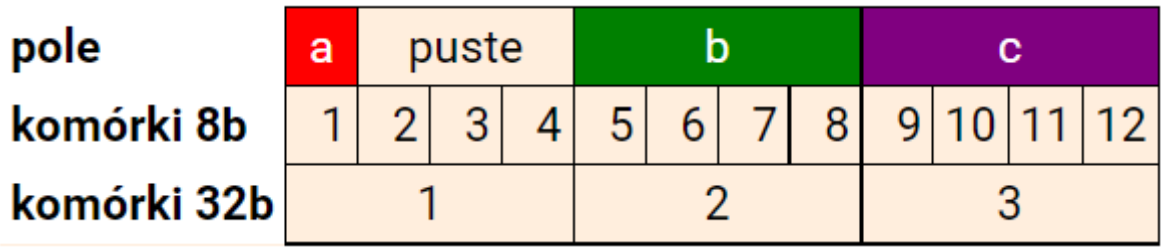
```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main() {
    struct {
        char a;
        int b;
        float c;
    } s;

    union {
        char a;
        int b;
        float c;
    } u;

    setlocale(LC_ALL, "");

    printf("Rozmiar struktury: %2d B\n"
           "Rozmiar unii : %2d B\n",
           sizeof(s), sizeof(u));

    return 0;
}
```

Przeanalizujmy to. Zmienna ***s*** jest strukturą o trzech polach: ***a*** typu ***char*** (1 bajt), ***b*** typu ***int*** (4 bajty) i ***c*** typu ***float*** (4 bajty). Jeśli zsumujesz rozmiary tych pól, otrzymasz 1 + 4 + 4 = 9 bajtów. Tymczasem program pokazuje 12 bajtów. Błąd? Nie, po prostu dane są

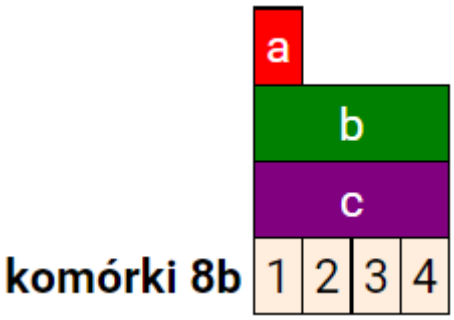
układane w pamięci tak, aby mikroprocesor Pentium szybko je pobierał. W architekturze 32-bitowej pamięć jest widziana przez mikroprocesor jako komórki 32-bitowe, które mogą dzielić się na dwie dane 16-bitowe lub na 4 dane 8-bitowe:



Komórki 32-bitowe posiadają adresy podzielne przez 4. Komórki 16-bitowe posiadają adresy podzielne przez 2. Takie rozwiązanie pozwala mikroprocesorowi pobierać dane w jednym cyklu dostępu do pamięci.
Typy ***int*** oraz ***float*** posiadają rozmiar 4 bajtów i są umieszczane w pamięci w komórkach o adresach podzielnych przez 4. Dlatego w strukturach mogą występować puste miejsca (niezajęte przez dane). Dla przykładu, struktura s z naszego programu jest rozmieszczona w pamięci w sposób następujący:



Teraz rachunek nam się zgodzi. Ponieważ pierwsze pole ***a*** jest typu ***char***, to zajmuje w pamięci tylko jedną komórkę 8-bitową. Drugie pole ***b*** jest typu ***int*** i zajmuje jedną komórkę 32-bitową. Z tego powodu nie może być umieszczone zaraz za polem ***a***, ponieważ się nie zmieści. Umieszczone zostaje pod adresem podzielnym przez 4. Za polem a powstają trzy niewykorzystywane w strukturze komórki 8-bitowe. Dlatego cała struktura zajmuje 1 + 3 + 4 + 4 = 12 bajtów.
A teraz unia: otrzymujemy wynik 4, ponieważ jest to maksymalny rozmiar danych, które w unii będą przechowywane:



Unia przechowuje tylko zawartość jednego pola. Dostęp do pól odbywa się identycznie jak w strukturze: za pomocą operatora kropka.

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main() {
    union {
        char a;
        int b;
        float c;
    } u;

    setlocale(LC_ALL, "");
    u.c = 3.141592;
    printf("Pole a = %c\n"
           "Pole b = %d\n"
           "Pole c = %f\n\n", u.a, u.b, u.c);
    u.b = 1234567890;
    printf("Pole a = %c\n"
           "Pole b = %d\n"
           "Pole c = %f\n\n", u.a, u.b, u.c);
    u.a = 'A';
    printf("Pole a = %c\n"
           "Pole b = %d\n"
           "Pole c = %f\n\n", u.a, u.b, u.c);

    return 0;
}
```

Program tworzy unię zawierająca trzy pola. Następnie wpisuje dane do kolejnych pól i wyświetla zawartość unii. Zwróć uwagę, że poprawnie będzie wyświetlone tylko to pole, do którego wprowadzono dane. Odczyt pozostałych pól daje jakieś przypadkowe wyniki.

Zadania do wykonania

1. Napisz strukturę ***trojkat*** przechowującą długości boków trójkąta. Napisz funkcję, która otrzymuje jako argument zmienną typu ***struct trojkat*** i zwraca jako wartość obwód trójkąta o przekazanego w argumencie.
2. Napisz funkcję, otrzymującą jako argumenty zmienną ***troj1*** typu ***struct trojkat*** z zadania 1 oraz zmienną ***troj2*** wskaźnik na zmienną typu ***struct trojkat***, i przepisuje zawartość zmiennej ***troj1*** do zmiennej wskazywanej przez ***troj2***.
3. Zdefiniuj strukturę ***punkt*** służącą do przechowywania współrzędnych punktów w trójwymiarowej przestrzeni kartezjańskiej. Napisz funkcję, która otrzymuje jako argumenty tablicę ***tab*** o argumentach typu ***struct punkt*** oraz jej rozmiar, i zwraca jako wartość najmniejszą spośród odległości pomiędzy punktami przechowywanymi w tablicy ***tab***. Zakładamy, że otrzymana w argumencie tablica ma co najmniej dwa argumenty.
4. Napisz funkcję, która otrzymuje jako argumenty tablice ***tab1*** i ***tab2*** o argumentach typu ***struct*** punkt zdefiniowanego w rozwiązaniu zadania 3 oraz ich rozmiar, i przepisuje zwartość tablicy ***tab1*** do tablicy ***tab2***.

4.5 (*)Zdefiniuj strukturę, która zawiera dwa pola: liczbowe i napisowe. Następnie w programie głównym stwórz:
   a) statyczną, 10-elementową tablicę takich struktur,
   b) dynamiczną, 10-elementową tablicę takich struktur,
   c) statyczną, 10-elementową tablicę wskaźników na struktury a następnie dla każdej z tych tablic w pętli wpisz wartości w pole liczbowe i napisowe, na końcu obliczając sumę dla pól liczbowych każdej z 3 tablic.

5. Zdefiniuj strukturę ***punkt10*** służącą do przechowywania współrzędnych punktów w dziesięciowymiarowej przestrzeni kartezjańskiej. Do przechowywania poszczególnych wymiarów wykorzystaj tablicę dziesięcioelementową. Napisz funkcję, która otrzymuje jako argumenty tablice ***tab1*** i ***tab2*** typu ***struct punkt10*** oraz ich wspólny rozmiar, i przepisuje zwartość tablicy ***tab1*** do tablicy ***tab2***.
6. (*)Zdefiniuj strukturę ***punktn*** służącą do przechowywania współrzędnych punktów w ***n***-wymiarowej przestrzeni kartezjańskiej. Do przechowywania poszczególnych wymiarów wykorzystaj tablicę n-elementową. W strukturze ***punktn*** przechowuj także ilość wymiarów przestrzeni. Napisz funkcję, która otrzymuje jako argumenty tablice ***tab1*** i ***tab2*** o argumentach typu ***struct punktn*** oraz ich wspólny rozmiar, i przepisuje zwartość tablicy ***tab1*** do tablicy ***tab2***. Zakładamy, że tablica ***tab2*** jest pusta (czyli nie musimy się martwić o jej poprzednią zawartość).
7. Zdefiniuj strukturę zespolone służącą do przechowywania liczb zespolonych. Zdefiniowana struktura powinna zawierać pola ***im*** i ***re*** typu ***double*** służące do przechowywania odpowiednio części urojonej i rzeczywistej liczby zespolonej. Napisz funkcję dodaj, która dostaje dwa argumenty typu zespolone i zwraca jako wartość ich sumę.
8. Zdefiniuj strukturę figura przechowującą wymiary ***fgur*** geometrycznych niezbędne do obliczenia pola. Struktura powinna mieć możliwość przechowywania wymiarów takich ***fgur***, jak: trójkąt, prostokąt, równoległobok i trapez. Rodzaj przechowywanej ***fgury*** powinien być zakodowany w wartości pola ***fig*** typu ***int***. Napisz funkcję pole, która dostaje jako argument zmienną ***f*** typu ***struct*** figura i zwraca jako wartość pole ***fgury*** której wymiary przechowuje zmienna ***f***.
9. Zdefiniuj strukturę ***punkt3D*** służącą do przechowywania współrzędnych zmiennoprzecinkowych ***x***, ***y*** i ***z***. Napisz funkcję, która dostaje jako argumenty wskaźniki do dwóch struktur ***punkt3D***, a następnie tworzy nową strukturę ***punkt3D***, która zawiera większe z wartości każdej współrzędnej i zwraca wskaźnik do niej.
   Przykład:
   We: pkt1 = {x:3, y:-5.5, z:34.3}, pkt2 = {x:2.5, y:-3, z:20}
   Wy: pkt3 = {x:3, y:-3, z:34.3}
10. Zdefiniuj strukturę ***dane_osobowe*** służącą do przechowywania imienia, nazwiska i wieku. Napisz funkcję ***najmlodszy***, która dostaje jako argument tablicę ***tab*** struktur ***dane_osobowe*** oraz jej rozmiar i wypisuje na standardowym wyjściu imię i nazwisko najmłodszej spośród osób, której dane przechowywane są w tablicy ***tab***.
11. Dana jest struktura określająca osobę:

```
typedef struct osoba {
    char nazwisko[50];
    int plec; //0-kobieta, 1-mezczyzna
    int wiek;
} osoba;
```

Napisz funkcję, która jako argument przyjmuje tablicę takich struktur oraz jej długość. Funkcja powinna ustalić średni wiek analizowanych osób, a następnie wyświetlić nazwisko osoby będącej w wieku najbliższym obliczonej średniej.

12. (*)Zdefiniuj strukturę o nazwie ***Student*** która zawiera 2 pola: dynamicznie zaalokowaną jednowymiarową tablicę elementów typu ***int*** przechowującą oceny danego studenta - oceny (tablica jest reprezentowana przez wskaźnik) oraz liczbę całkowitą ***ilosc***, wskazującą ilość ocen danego studenta. Napisz funkcję, która dostaje 2 argumenty: tablicę struktur ***Student*** o nazwie ***tab*** oraz liczbę całkowitą ***n*** wskazującą na ilość studentów. Tablica ***tab*** przechowuje dane o ocenach ***n***-kandydatów na studia. Indeksy elementów tablicy ***tab*** są jednocześnie numerami kandydatów. Na studia przyjmowane są osoby, których średnia ocen jest większa lub równa 4.0. Funkcja powinna wyświetlić listę osób przyjętych na studia (numer kandydata oraz średnią jego ocen).
    Przykład:
    We: n = 3, ***tab[0]*** : ***ilosc***: 2, ***oceny***: [3, 4], ***tab[1]*** : ***ilosc***: 4, ***oceny***: [3, 4, 5, 4], ***tab[2]*** : ***ilosc***: 1, ***oceny***: [5],
    Wy: 1 - 4.0, 2 - 5.0
13. (*)Zadeklaruj typ strukturalny do reprezentowania informacji o wynikach uruchomień pewnych programów: czas startu i czas zakończenia w sekundach licząc od pewnego ustalonego momentu. Napisz funkcję, która przyjmuje tablicę takich struktur i zwraca liczbę, która oznacza, ile pełnych wykonań programu działającego najkrócej zdążyłoby się zrealizować w czasie działania programu wykonującego się najdłużej.
14. Zdefiniuj strukturę ***planeta***, posiadającą: numer planety (liczba całkowita), nazwa ***planety*** (napis 50-literowy), promień planety (liczba zmiennoprzecinkowa). Napisz funkcję przyjmującą tablicę planet oraz jej rozmiar. Funkcja ma zwrócić planetę (całą strukturę) o największej objętości ($V = \frac{4}{3} \cdot 3,14 \cdot r^3$). Okazuje się jednak, że planety o numerach nieparzystych nie są idealnymi kulami, dlatego w ich przypadku objętość należy pomniejszyć o 10% w stosunku do wyniku uzyskanego ze wzoru.

Zagadnienia i zadania oznaczone (*) są dla chętnych.