C - 8 - Struktury

8.1 Definicja i przykłady

Podczas omawiania tablic pojawił się program z planetami (nazwy, odległości od słońca, długość roku słonecznego). Przechowywaliśmy te dane w trzech tablicach typu **char***, **int** oraz **double**. O wiele wygodniejszym sposobem jest przechowywanie takich danych w strukturach. Przykładową strukturę zadeklarujemy jako:

```
struct planeta{
        char nazwa[20];
        int odleglosc;
        double rok;
};
```

Przykład użycia struktury w programie – przy okazji widzimy jak można przekazać strukturę do funkcji :)

```
#include "stdafx.h"
#include "string.h"

struct planeta{
        char nazwa[20];
        int odleglosc;
        double rok;
};

void wydrukujPlanete(struct planeta p){
        printf("\nNAZWA: %s\n",p.nazwa);
        printf("ODLEGLOSC OD SLONCA: %d\n",p.odleglosc);
        printf("ROK SLONECZNY WZGLEDEM ZIEMSKIEGO: %.4lf\n",p.rok);
}

int main(int argc, char* argv[]){
        struct planeta p1;
        struct planeta p2;

        strcpy(p1.nazwa,"Merkury");
        p1.odleglosc = 58;
        p1.rok = 0.2408;

        strcpy(p2.nazwa,"Wenus");
        p2.odleglosc = 108;
        p2.rok = 0.6152;

        wydrukujPlanete(p1);
        wydrukujPlanete(p2);

        return 0;
}
```

Struktury definiujemy z reguły pod includami.

Średnik, średnik, średnik !!!

Trzeba wiedzieć, co jest czym:
struct planeta jest nazwą typu
p jest nazwą zmiennej

Prawie wszyscy studenci piszą:
p1.nazwa = "Merkury";
i dziwią się, że nie działa :(

NAZWY PRZYPISUJEMY ZA POMOCĄ strcpy !!!!!!

Zaraz nad **main**em wpisz funkcję przypisującą dane do struktury:

```
void przekazDane (struct planeta *gdzie,char n[20],int o,double r){
        strcpy(gdzie->nazwa,n);
        gdzie->odleglosc = o;
        gdzie->rok = r;
}
```

Tutaj po nazwie struktury stosujemy strzałkę -> ponieważ mamy do czynienia ze wskaźnikiem do struktury. W funkcji wyświetlającej nie było wskaźnika, więc stosowaliśmy kropkę. Warto więc zapamiętać: jeżeli gwiazdka, to strzałka.

i zastąp trzy linijki przypisujące wartości do planety **p1** przez jedną linijkę:

```
        przekazDane (&p1,"Merkury",58,0.2408);
```

Analogicznie przekaż dane do drugiej planety. Sprawdź, czy program dobrze działa.

Struktury można przechowywać w tablicach!!! Przekopiuj plik **planety.txt** do katalogu ze swoim projektem i zaraz nad funkcją **main** wpisz nową funkcję:

```
void wczytajDane(char *nazwaPliku,struct planeta tablica[9]){
        FILE *plik = fopen(nazwaPliku,"r");

        if(plik==NULL){
                printf("Blad odczytu z pliku %s",nazwaPliku);
                return;
        }

        char pom_n[20];
        int pom_o;
        double pom_r;
        int licznik=0;
        while(!feof(plik)){
```

```
                fscanf(plik,"%s %d %lf",pom_n,&pom_o,&pom_r);
                przekazDane(&tablica[licznik++],pom_n,pom_o,pom_r);
        }
        fclose(plik);
}
```

Dotychczasową zawartość funkcji **main** zastąp przez:

```
        struct planeta tablicaPlanet[9];
        wczytajDane("planety.txt",tablicaPlanet);
        for(int i=0;i<9;i++) wydrukujPlanete(tablicaPlanet[i]);
        return 0;
```

i uruchom program.

Zadanie 1. Napisz funkcję **zapiszDoPliku(char *nazwaPliku,struct planeta tablica[9])**, która zapisze do pliku planety w odwrotnej kolejności.

Struktura może też zawierać inne struktury, np:

```
#include "stdafx.h"
#include "math.h"

struct punkt{
        int x;
        int y;
};

struct kwadrat{
        struct punkt lewyGorny;
        struct punkt prawyDolny;
};

int obliczPole(struct kwadrat fig){
        return abs(fig.lewyGorny.x - fig.prawyDolny.x)*abs(fig.lewyGorny.y - fig.prawyDolny.y);
}

int main(int argc, char* argv[]){
        struct kwadrat figurka;
        figurka.lewyGorny.x = 2;
        figurka.lewyGorny.y = 4;
        figurka.prawyDolny.x = 6;
        figurka.prawyDolny.y = 1;

        printf("Pole kwadratu wynosi %d\n",obliczPole(figurka));
        return 0;
}
```

Zadanie 2. Przerób program z planetami - pod strukturą definiującą planetę dodaj definicję struktury:

```
struct ukladSloneczny{
        struct planeta tablicaPlanet[9];
        char nazwa[20];
};
```

Wczytaj z dwóch plików dane dwóch różnych układów słonecznych (ten drugi może być zmyślony), a następnie wyświetl informacje o nich w jakiejś zwięzłej tabelce.

Większy program na weekend:

Napisz program, który zawiera dwa typy struktur:

1) struktury przechowujące dane o samochodzie (patrz: plik **samochody.txt**), a więc: marka, pojemność silnika, rok produkcji, przebieg, cena, województwo sprzedającego
2) strukturę przechowującą dane o salonie samochodowym – adres, imię i nazwisko właściciela oraz tablicę struktur z ofertą samochodów

W programie zaimplementuj funkcje, które:

- wyświetlają informacje o pojedyńczym samochodzie
- wyświetlają informację o salonie samochodowym
- sortują i wyświetlają samochody według ceny (bąbelkowo)
    - należy samemu poeksperymentować z zamianą struktur w tablicy !!!
- wypisują samochody pochodzące z województwa pomorskiego

Dodaj do pliku i do struktury pole boolowskie oznaczające, czy samochód brał udział w wypadku, czy nie. Przepisz funkcję wyświetlającą tak, by uwzględniała nowe dane. Dopisz funkcję wypisującą tylko samochody bezwypadkowe.

8.2 Ciekawy przykład dla liczb zespolonych

Poniższy przykład ilustruje zastosowanie struktur w matematyce:

```
#include "stdafx.h"
#include "math.h"

struct zespolona{
        double re;
        double im;
};

struct zespolona* dodaj(struct zespolona a,struct zespolona b){
        struct zespolona *wynik = new struct zespolona;
        wynik->im = a.im+b.im;
        wynik->re = a.re+b.re;
        return wynik;
}

void wyswietl(struct zespolona z){
        if(z.im==0 && z.re==0) printf("0\n");
        else if(z.re==0) printf("%.2lfi",z.im);
        else if(z.im==0) printf("%.2lf",z.re);
        else if(z.im>0) printf("%.2lf+%.2lfi",z.re,z.im);
        else printf("%.2lf%.2lfi",z.re,z.im);
}

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

        struct zespolona z1={0.5,2.2};
        struct zespolona z2={-4.0,3.0};
        struct zespolona *w = dodaj(z1,z2);

        wyswietl(z1);
        printf(" + ");
        wyswietl(z2);
        printf(" = ");
        wyswietl(*w);
        delete(w);
        return 0;
}
```

Tu mamy przykład funkcji zwracającej wskaźnik.

Tak też można przypisywać wartości do struktury.

Czyścimy pamięć po wskaźniku.

Zadanie 3. Dopisz funkcję mnożące i dzielące dwie liczby zespolone. Dopisz też funkcję typu **double**, która będzie obliczać moduł z liczby zespolonej.

Zadanie 4. Przerób program tak, by liczby zespolone były przechowywane w tablicy struktur (co najmniej 10 liczb zespolonych). W pętli wykonaj na liczbach zespolonych dodawanie lub inne operacje, wyświetl zawartość tablicy po tych operacjach.

Zadanie 5. Sprawdź ten sposób przypisywania wartości w programie z planetami.

8.3 Tworzenie własnych typów: *typefdef* i *enum*

Aby zadeklarować strukturę, w dotychczasowych przykładach trzeba było użyć aż dwóch wyrazów: **struct** oraz nazwy struktury. W programie dla liczb zespolonych zamień definicję struktury na następującą:

```
typedef struct{
        double re;
        double im;
} zespolona;
```

typedef to skrót od „type definition”

Następnie wykasuj wszystkie słowa **struct** z reszty programu. Zamiast kasować je ręcznie naciśnij **Ctrl+h** – pojawi się małe okienko o tytule „Replace”. W polu „Find what” wpisz słowo **struct**, pole „Replace with” zostaw puste. Naciśnij przycisk „Replace All”. Upewnij się, że nie skasowałeś słowa **struct** po słowie **typedef** i uruchom program.

Zadanie 6. Napisz program ze strukturą **czas**, która zawiera trzy pola typu **int**: godzina, minuta, sekunda. Napisz funkcję **dodawaj**, która pobiera dwie zmienne czasowe, dodaje w odpowiedni sposób czasy i zwraca wskaźnik do nowej struktury. Użyj słowa **typedef**.

Innym sposobem tworzenia własnego typu jest enumeracja – lista stałych, które muszą być wartościami typu **int**. Przykład:

```
#include "stdafx.h"
#include "math.h"

enum przedmioty  {JEZ_PRO, ALG_I_STR_DAN, ALG_PRO, MET_PRO, PAK, KRY };

char* napis(przedmioty p){
        if(p==JEZ_PRO) return "jezyki programowania";
        if(p==ALG_I_STR_DAN) return "algorytmy i struktury danych";
        if(p==ALG_PRO) return "algorytmy probabilistyczne";
        if(p==MET_PRO) return "metody programowania";
        if(p==PAK) return "pakiety matematyczne";
```

Tu nie ma = !!!

```
        if(p==KRY) return "kryptografia";
        return "------";
}

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

        int poniedzialek[6][2]={    {202,JEZ_PRO},
                                    {202,ALG_I_STR_DAN},
                                    {460,ALG_PRO},
                                    {13, MET_PRO},
                                    {13, PAK},
                                    {265,KRY}};

        printf("sala    przedmiot\n");
        for(int i=0;i<6;i++)
                printf("%d\t%s\n",poniedzialek[i][0],napis((przedmioty)poniedzialek[i][1]));

        return 0;
}
```

Rzutujemy na nasz nowy

Enumeracje przydają się, gdy mamy do czynienia z ograniczoną liczbą danych (np. nazwy przedmiotów), które chcemy przechowywać w tablicach razem z wartościami typu **int** (np. numery sal, kod prowadzącego). Można oczywiście używać liczby 0 zamiast języków programowania, 1 zamiast algorytmów i struktur danych. Jednakże po otwarciu programu parę miesięcy później nikt już nie będzie pamiętał, że pakiety matematyczne miały akurat numer 4. Zamiast enumeracji można w powyższym przykładzie użyć sześciu linijek **#define ...** ale byłoby to niewygodne.

Jeżeli nie chcemy, by enumeracja zaczynała się od zera, lecz np. od dwójki, to możemy napisać:
```
enum przedmioty  {JEZ_PRO=2, ALG_I_STR_DAN, ALG_PRO, MET_PRO, PAK, KRY };
```
Kolejne przedmioty będą wówczas miały numery 3,4,5,6 oraz 7.

Możemy też zażądać, by każdy element enumeracji miał zdefiniowaną wartość, np:
```
enum rozmiary {XS=36, S=38, M=40, L=42, XL=44};
```

Zadanie 7. Napisz program z dwiema enumeracjami – sam wymyśl, czego będą dotyczyć. Następnie umieść w dwukolumnowej tablicy typu **int** kilka takich wartości. Napisz funkcję, która będzie wyświetlać zawartość tablicy w zrozumiały sposób.

8.4 Najczęstsze błędy

Zapominanie o **średniku** po definicji struktury.
Pisanie znaku **=** przed nawiasem klamrowym enumeracji.
Zapominanie, że jeżeli mamy wskaźnik do struktury, to odwołujemy się do jej pól poprzez strzałkę.
Niekorzystanie z **typedef**ów, przez co kod programu wydłuża się niepotrzebnie.
Nieczyszczenie pamięci po wskaźnikach do struktur.

8.5 Quiz

1. Która z poniższych możliwości jest poprawnym odwołaniem do pola **var** w zmiennej strukturowej **b**?
   a) b->var b) b.var c) b-var d) b>var

2. Która z poniższych możliwości jest poprawnym odwołaniem do pola **var** we wskaźniku **b** do zmiennej strukturowej?
   a) b->var b) b.var c) b-var d) b>var

3. Który z poniższych zapisów jest poprawną definicją struktury o jednym polu?
   a) struct {int a;}
   b) struct mojaStruktura {int a;}
   c) struct mojaStruktura int a;
   d) struct mojaStruktura {int a;};

4. Która z poniższych instrukcji prawidłowo deklaruje zmienną **x** typu strukturowego?
   a) struct moja;
   b) struct moja x;
   c) moja;
   d) int x;