Struktury

Motivační příklad

- chceme v programu zpracovávat informace o bodech v rovině
  - každý bod má souřadnice [x, y]
  - naprogramujeme funkci, která vrací vzdálenost bodu od počátku

$$d = \sqrt{x^2 + y^2}$$

```
float vzdalenost(float x, float y)
{
  return sqrt(x*x+y*y);
}

int main()
{
  float xA,yA; // bod A
  float xB,yB; // bod B
  float d;
  xA = 3.5; yA = 2;
  d = vzdalenost(xA,yA);
}
```

Vadí nám na tom něco?

- bod chápeme jako dvojici souřadnic, ale toto „seskupení souřadnic“ ze zápisu programu nevidíme
  - logicky by bylo vhodné mít dvě proměnné pro dva body, nikoliv čtyři proměnné
  - pokud bychom chtěli ukládat souřadnice více bodů do pole, musíme deklarovat dvě pole, jedno pro x-ové, druhé pro y-ové souřadnice

```
float X[20];
float Y[20];
```

Strukturované datové typy

- struktura je heterogenní datový typ
  - proměnná typu struktura tedy obsahuje několik položek různých datových typů, které ovšem spolu logicky souvisejí
  - struktura umožňuje „společné“ pojmenování všech sdružených údajů, s nimiž se potom pohodlněji pracuje

Proměnná strukturovaného datového typu

- deklarace proměnné bod typu struktura, která reprezentuje body v dvourozměrném prostoru:

```
struct {
   float x;
   float y;
} bod;
```

- strukturovaný datový typ nemůže být použit jinde, protože není pojmenovaný

- pojmenovaná struktura:

```
struct Bod {
  float x;
  float y;
} bod;
```

- deklarace nových proměnných `b1, b2:`

```
struct Bod b1, b2;
```

- pojmenovaná struktura může být použita také jako parametr funkcí, ale klíčové slovo **`struct`** musí být uvedeno

- deklarace nového typu `Bod`:

```
typedef struct {
 float x;
 float y;
} Bod;
```

- deklarace proměnné `bod1`, která je typu `Bod`:

```
Bod bod1;
```

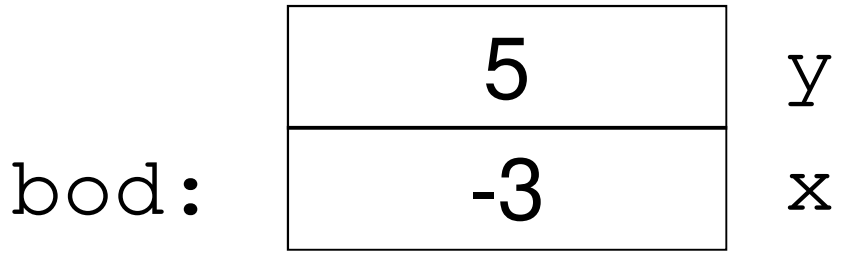
- deklarace pole „bodů“

```
Bod body[20];
```

- přístup k položkám se provádí pomocí tečkové notace:

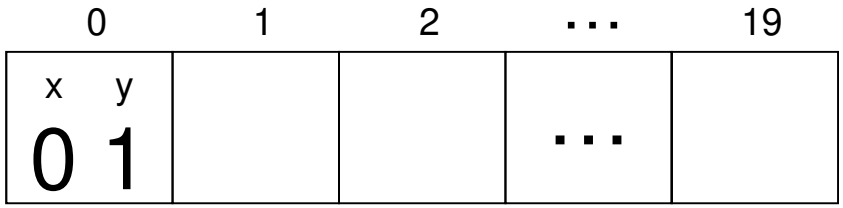
```
bod.x = -3; bod.y = 5;
bod1.x = 2; bod1.y = 0;
body[0].x= 0; body[0].y = 1;
```

- uložení v paměti:



- pole struktur `Bod body[20];`
  - uložení v paměti

`body:`



- stejně jako u pole je možné již při deklaraci inicializovat položky struktury (tzv. konstruktor):

```
Bod b1 = {3,-1};
```

Ukazatelé na strukturu

- deklarace

```
Bod *pb;
```

- dynamická alokace jedné struktury (pole délky 1)
  - v C

```
pb = (Bod*)malloc(sizeof(Bod));
```

  - v C++

```
pb = new Bod;
```

- přístup k položkám

```
*pb.x = 5;
```
nebo **pb->x**

Ukazatelé na strukturu

- dynamická alokace pole o velikosti *n*
  - v C

```
pb = (Bod*)malloc(sizeof(Bod)*n);
```

  - v C++

```
pb = new Bod[n];
```

- přístup

`pb[1].x = 5;` nebo `*(pb+1).x` nebo `(pb+1)->x`

Předávání struktur funkcím jako parametr

- definice funkce

```
float vzdalenost(Bod p)
{
 return sqrt(p.x*p.x+p.y*p.y);
}
```

- volání

```
vzdalenost(p1);
```

- neefektivní, celá struktura (všechny položky) jsou kopírovány na zásobník

Předávání struktur funkcím jako parametr

- ukazatel jako parametr

```
float vzdalenost(Bod *p)
{
 return sqrt(p->x*p->x+p->y*p->y);
}
```

- volání

```
vzdalenost(&p1);
```

- na zásobník je předána pouze adresa struktury

Předávání struktur funkcím jako parametr

- reference jako parametr

```
float vzdalenost(Bod &p)
{
 return sqrt(p.x*p.x+p.y*p.y);
}
```

- volání

```
vzdalenost(p1);
```

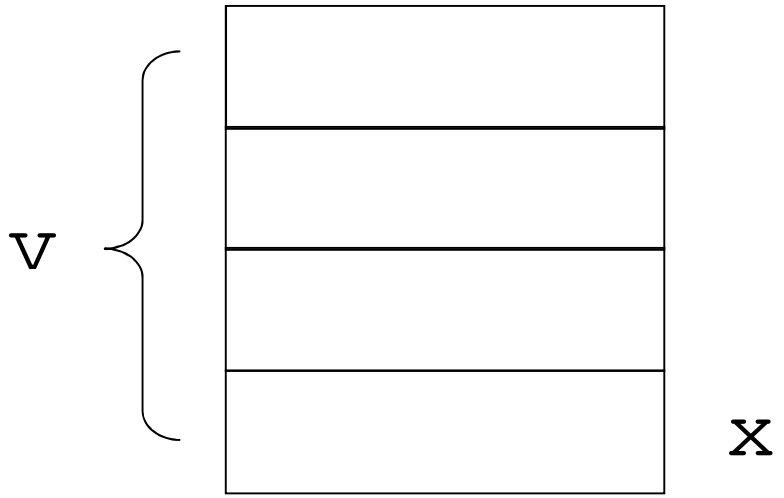
- na zásobník je předána pouze adresa struktury

Poznámka

- **struktura** odpovídá záznamu (record) v Pascalu
- existuje ještě datový typ **sjednocení (union)**
  - analogie variantního záznamu v Pascalu

Poznámka

```
union {
   char x;
   int v;
} uv;
```



Jiný příklad

```
typedef struct
{
   char znacka_vozidla[30];
   char RZ[10];
   int objem_valcu;
} Auto;

Auto auto1;
```

- přístup k položkám:

```
auto1.objem_valcu = 1221;
strcpy(auto1.znacka_vozidla,"Škoda");
strcpy(auto1.RZ,"1A1 01 01");
```

- v klasickém C musíme je řetězec reprezentován pole znaků, nelze uložit text do pole pomocí přiřazení, musíme použít knihovní funkci `strcpy`

- naštěstí máme v C++ typ **`string`**

```
typedef struct
{
  string znacka_vozidla;
  string RZ;
  int objem_valcu;
} TAuto;

TAuto auto1;

auto1.objem_valcu = 1221;
auto1.znacka_vozidla = "Škoda";
auto1.RZ = "1A1 01 01";
```

Uložení data pomocí struktury

```
typedef struct
{
   int den;
   int mesic;
   int rok;
} Datum;

Datum datum_narozeni;
datum_narozeni.den = 5;
datum_narozeni.mesic = 12;
```

Položkou struktury může být struktura

```
typedef struct
{
   string jmeno;
   string prijmeni;
   Datum dat_nar;
   ...
} Osoba;
```

Položkou struktury může být struktura

```
Osoba student;
student.jmeno = "Vit";
student.dat_nar.den = 5;
student.dat_nar.mesic = 5;
```

Poznámka

- pokud bychom chtěli ukládat informace o několika osobách do pole a nevyužívali strukturovaný typ:

```
string jmeno[20];
string prijmeni[20];
int den[20];
int mesic[20];
int rok[20];
```

Poznámka

- při deklaraci typu `Osoba` (viz snímek 17) stačí jedno pole struktur:

```
Osoba lide[20];
lide[0].jmeno = "Josef";
lide[0].prijmeni = "Novak";
lide[0].dat_nar.den = 19;
```

Příklad – formát BMP

- grafický formát pro uložení obrázků bmp (Bitmap) obsahuje úvodní hlavičku o délce 14 bytů.

Položka	Délka položky	Popis
typ	2 byty	2 znaky „BM“
velikost	4 byty	celková velikost souboru
rezervováno1	2 byty	rezervováno pro pozdější použití, musí být na 0
rezervováno2	2 byty	rezervováno pro pozdější použití, musí být na 0
posun	4 byty	posun obrazových dat od začátku této hlavičky (struktury)

- předpokládejme, že pracujeme na platformě Intel
  - uložení dat little-endian, tj. slabiky nižšího řádu jsou uloženy na nižších adresách),
  - `sizeof(unsigned short) == 2`
  - `sizeof(unsigned int) == 4`

- strukturu, která odpovídá hlavičce souboru, nadeklarujeme následovně:

```
typedef struct
{
 unsigned char B;
 unsigned char M;
 unsigned int velikost;
 unsigned short res1;
 unsigned short res2;
 unsigned int posun;
}  THeadBMP;
```

- nadeklarujeme proměnnou pro hlavičku:

```
THeadBMP hlavicka;
```

- hlavičku načteme ze soubor vst, který byl otevřen pomocí fopen v *binárním* módu:

```
fread((void*)&hlavicka,sizeof(THeadBMP),1,vst)
```

- *upozornění:*
  - některé překladače v rámci optimalizace přístupu do paměti neukládají položky struktur těsně za sebou, ale zarovnávají je na adresy dělitelné 4. V takovém případě by mezi položkami M a velikost byla v paměti mezera a velikost struktury by byla větší než velikost hlavičky. Museli bychom tedy v překladači tuto optimalizaci (zarovnání položek na adresy dělitelné 4) vypnout. Např. v gcc použijeme přepínač překladu -fpack-struct=1, popř. lze použít makro #pragma pack(1) (zarovnat na 1 byte - pack(1) je makro z důvodu kompatibility s překladači Microsoft)