Structures

Motivation

- we want to process points in 2D space in the sense of analytic geometry
  - each point is represented by coordinates [x, y]
  - we define the function that calculates the distance of the point from the origin

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

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

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

It is not so elegant

- the point is understood as a couple of coordinates x, y, but "this couple" is not evident from the source code
  - logically: two points – two variables instead of four ones
  - if we want to store point into array we must define two arrays for each coordinate

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

Structured data type

- structured data type is heterogeneous data type
  - the variable of the structured data type is a collection of several items of different types
    - elements are logically related
  - structures allow "common naming" of several items

Variable of Structured Data Type

- declaration of the variable point of the structured data type that represents point in 2D space:

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

- structured data type can’t be used in other place of the code because it has no name

- named structure:

```
struct Point {
  float x;
  float y;
} point;
```

- declaration of new variables `p2, p2`:

```
struct Point p1, p2;
```

- the named structure can be used as the type of the parameter in functions, but the keyword **`struct`** must be always written

- declaration of new user - type `Point`:

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

- declaration of the variable `point1` that is of the type `Point`:

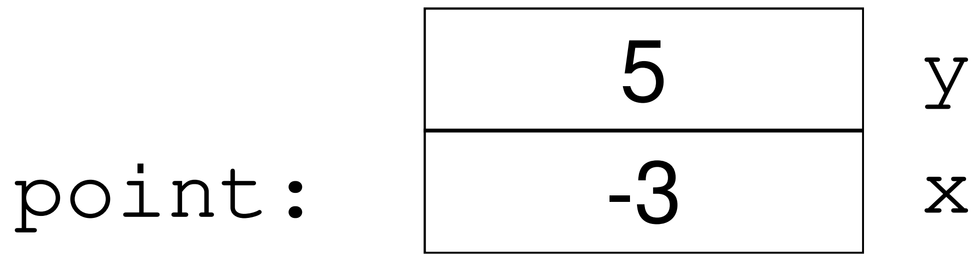
```
Point point1;
```

- declaration of the array:
  `Point points[20];`

- access to the elements: using dot notation:

```
point.x = -3; point.y = 5;
point1.x = 2; point1.y = 0;
points[0].x= 0; points[0].y = 1;
```

- storage structure in memory:

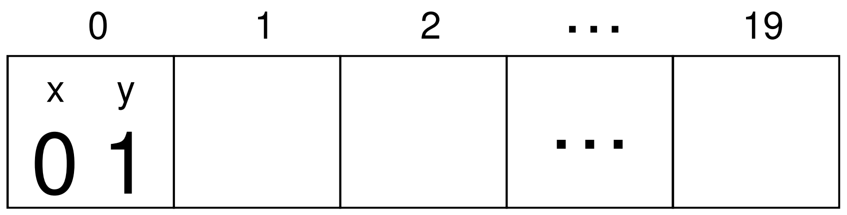


- array of the structures

```
Point points[20];
```

  – storage in memory

```
points:
```



- items can be initialized within declaration (by constructor):

```
Point p1 = {3,-1};
```

Pointers to structures

- declaration

```
Point *pp;
```

- dynamic allocation of one structure (array of the length 1)
  - in C

```
pp = (Point*)malloc(sizeof(Point));
```

  - in C++

```
pp = new Point;
```

- access

`*pp.x = 5;` or **`pp->x`**

Pointers to structures

- dynamic allocation of the array of the length *n*
  - in C

```
pp = (Point*)malloc(sizeof(Point)*n);
```

  - in C++

```
pp = new Point[n];
```

- access

`pp[1].x = 5;` or `*(pp+1).x` or `(pp+1)->x`

Passing structures to functions

- function definition

```
float distance(Point p)
{
 return sqrt(p.x*p.x+p.y*p.y);
}
```

- calling

```
    distance(p1);
```

- non-effective, whole structure (all elements) are copies to the stack

Passing structures to functions

- using pointers

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

- calling

```
    distance(&p1);
```

- only address is passed to the stack

Passing structures to functions

- using reference parameter

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

- calling

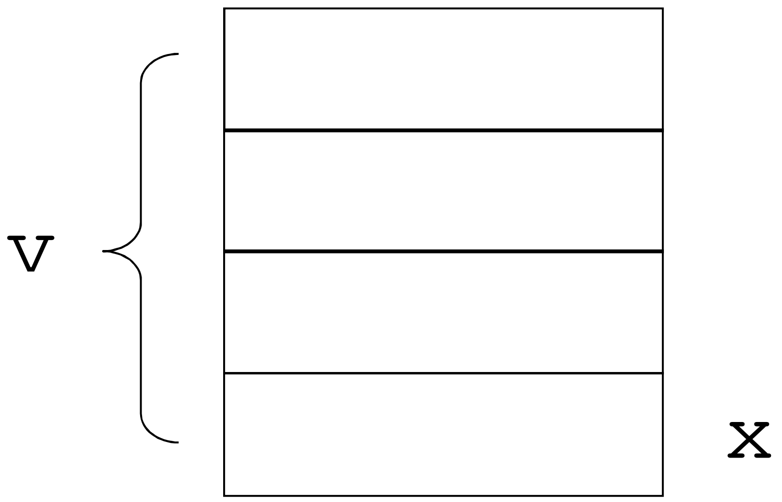
```
    distance(p1);
```

- internally only address is passed to the stack

Note

- other data type exists in C:
  union

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



Example

```
typedef struct
{
   char vehicle[30];
   char LN[10];
   int volume_of_cylinder;
} Car;

Car my_car;
```

- access to items:

```
my_car.volume_of_cylinder = 1221;
strcpy(my_car.vehicle,"VW");
strcpy(my_car.LN,"1A1 01 01");
```

- library function `strcpy` must be used in C to copy strings

- new type **`string`** is defined in C++

```
typedef struct
{
   string vehicle;
   string LN;
   int volume_of_cylinder;
} Car;

my_car.volume_of_cylinder = 1221;
my_car.vehicle = "VW";
my_car.LN = "1A1 01 01";
```

Store date using structure

```
typedef struct
{
   int day;
   int month;
   int year;
} Date;

Date birth_date;
birth_date.day = 5;
birth_date.month = 12;
```

The structure can be part of another structure

```
typedef struct
{
   string first_name;
   string surname;
   Date birth_date;
   ...
} Person;
```

The structure can be part of another structure

```
Person student;
student.name = "Vit";
student.birth_date.day = 5;
student.birth_date.month = 5;
```

Note

- we want to store data about people to array using structure …
- one array of structures is declared:

```
Person people[20];
people[0].first_name = "Martin";
lide[0].surname = "Smith";
lide[0].birth_date.day = 19;
```

Note

- if we had no structures we would declare five arrays:

```
string first_name[20];
string surname[20];
int day[20];
int month[20];
int year[20];
```

Example – BMP format

- graphical format to store pictures
- the file contains header of the length 14 bytes in the beginning

Item	**Size**	**Description**
*typ*e	2 bytes	2 characters „BM“
size	4 bytes	the total size of the file
*reserved*1	2 bytes	reserved for future use, must be set to 0
reserved2	2 bytes	reserved for future use, must be set to 0
offset	4 bytes	the offset, i.e. starting address, of the byte where the bitmap image data (pixel array) can be found in the file

- we suppose:
  - Intel platform
    - little-endian, i.e. the least significant bytes at lower addresses
  - **`sizeof`**`(`**`unsigned short`**`) == 2`
  - **`sizeof`**`(`**`unsigned int`**`) == 4`

- structure data type corresponding to the structure of the head:

```
typedef struct
{
 unsigned char B;
 unsigned char M;
 unsigned int size;
 unsigned short res1;
 unsigned short res2;
 unsigned int offset;
}  THeadBMP;
```

- declaration the variable to store header:

```
THeadBMP header;
```

- reading the header from the file `fi` which was opened in binary mode using `fopen`:

```
fread((void*)&header,sizeof(THeadBMP),1,fi)
```

- *important note:*
  - compilers optimize access to the memory (reading from bus) and items of the structures are not placed one after another but they are aligned to addresses divided by 2 or 4. In this case the gap of 2 bytes is between items M and size and the size of the whole structure is bigger than 14 bytes. When block of bytes is read from the file in this case data is not correctly in items. Packing structures must be switched on – for example by switch `-fpack-struct=1` in GCC, or using directive `#pragma pack(1)` (items alignment to 1 Byte – compatibility with Microsoft compilers)