

Структури

Пламен Танов
Ненко Табаков

Технологично училище „Електронни системи“
Технически университет – София

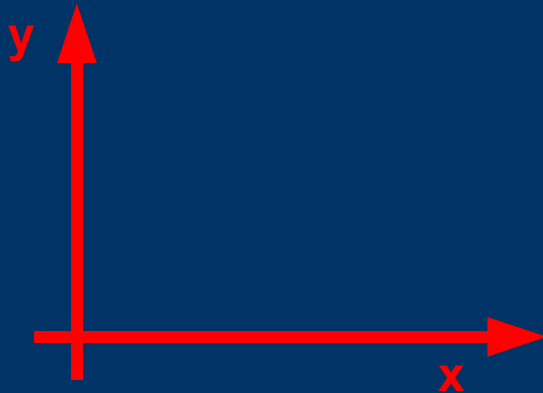
версия 0.1

Въведение

- Структурата представлява набор от една или повече променливи (които могат да бъдат от различни типове) групирани под общо име за по-лесна работа с тях
 - Структурите спомагат за организацията на по-сложни данни като дават възможност група от променливи, описващи даден предмет, да се приема за едно цяло
 - Структурата е като всяка друга променлива – може да присвоява стойността на друга структура от същия тип, да ѝ се взима адреса, да се предава и връща от функции и т.н.
-
-

Пример

Структура описваща точка от координатната система

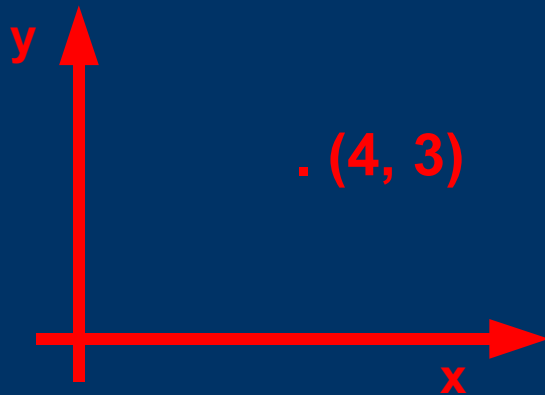


```
struct point {  
    int x;  
    int y;  
};
```



Пример

Структура описваща точка от координатната система



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



Действия със структури

- Деклариране
 - Инициализиране
 - Достъп до полетата на структура
 - Копиране на структури
 - Структури като аргумент и резултат на функции
 - Указатели към структури
 - Масиви от структури
-
-

Деклариране

```
struct име_на_типа {  
    променлива1;  
    променлива2;  
    ....  
};
```

```
struct point {  
    int x;  
    int y;  
}; //декларира тип
```

```
/*създаване на променлива в  
от тип struct point*/
```

```
struct point pt;
```

```
struct {  
    променлива1;  
    променлива2;  
    ....  
} изброяване на променливи;
```

```
struct {  
    int x;  
    int y;  
} a, b, c;
```

```
/*създава променливи a, b  
и c от тип struct */
```

Инициализация

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

```
//като масив – заградени м/у {} и отделени със запетая  
struct point pt = {320, 240};
```

```
//pt.x = 320;  
//pt.y = 240;
```

Достъп до променливите

име_на_структурата.член_променлива

```
struct point {  
    int x;  
    int y;  
};  
...  
  
struct point pt = {320, 240};  
  
printf ("x = %d, y = %d", pt.x, pt.y);
```

Вграждане на структури

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

```
struct rect {  
    struct point pt1;  
    struct point pt2;  
};  
...
```

//както се инициализира структура

//всеки елемент се инициализира поотделно като структура

```
struct rect screen = {{10, 20}, {30, 40}};
```

```
screen.pt1.x = 10;
```

```
screen.pt1.y = 20;
```

```
screen.pt2.x = 30;
```

```
screen.pt2.y = 40;
```

Копиране на структури

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

int main () {
    struct point a, b;
    a.x = 10;
    a.y = 20;
    b = a; //копират се стойностите на всички полета на a в b
    //или
    b.x = a.x;
    b.y = a.y;
    //b.x = 10; b.y = 20;
}
```

Структури като аргумент на функции

Структурите могат да се предават като аргументи на функции.

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

```
struct point addpoint (struct point p1, struct point p2) {  
    p1.x += p2.x; //работи се с копието, а не с оригинала!  
    p1.y += p2.y;  
    return p1;  
}
```

Структури като резултат на функции

Структурите могат да се връщат като резултат от функции.

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

```
struct point makepoint (int x, int y) {  
    struct point temp;  
    temp.x = x;  
    temp.y = y;  
    return temp;  
}
```

Пример

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

struct rect {
    struct point pt1;
    struct point pt2;
};

struct point makepoint (int x, int y);
struct point addpoint (struct point p1, struct point p2);

int main () {
    struct point result;
    struct rect screen;
    screen.pt1 = makepoint(10, 20);
    screen.pt2 = makepoint(30, 40);
    result = addpoint(screen.pt1, screen.pt2);
}
```

Указатели към структури₁

Указателите към структура са идентични на указателите към всички други променливи

`s struct point *pp; //pp е указател към структура от тип s struct point`

Пример

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

int main () {
    struct point pt = {20, 30};
    struct point *pp;
    pp = &pt;
    printf("X = %d, Y = %d\n", (*pp).x, (*pp).y);
}
```

Указатели към структури₂

При $(*pp) . x$ кръглите скоби са задължителни, защото приоритетът на оператора за достъп до член на структурата „.” е по-висок от приоритета на оператора „*“.

Друг начин за достигане до член на структурата (посредством указател) е чрез оператора „->“

`pp -> член_на_структурата`

Пример

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

int main () {
    struct point pt = {20, 30};
    struct point *pp;
    pp = &pt;
    printf("X = %d, Y = %d\n", pp->x, pp->y);
}
```

Указатели към структури₃

Операторите „.“, „->“, „()“ и „[]“ имат най-висок приоритети за извикване.

```
struct {  
    int len;  
    char *str;  
} *p;
```

```
++p->len; //еквивалентно на ++(p->len);  
        //ще увеличи стойността на len
```

```
*p->str; //еквивалентно на *(p->str);  
        //ще вземе стойността, към която сочи str
```

Масиви от структури

```
s struct key{  
    int a;  
    int square_a;  
};
```

```
s struct key square_roots [5]; //работи се като с обикновен масив
```

```
...  
for (i = 0; i<5; i++) {  
    square_roots [i].a = i;  
    square_roots [i].square_a = i * i;  
}
```

//или:

```
s struct key sqr [5] = { //както се инициализира масив от даден тип  
    {0, 0},           //както се инициализира структура  
    {1, 1},  
    {2, 4},  
    {3, 9},  
    {4, 16},  
};
```

*typedef*₁

Езикът С предоставя възможност за създаване на нови имена за вече съществуващи типове данни.

```
typedef int Length; //вместо #define Length int
```

```
Length len, max;  
Length *lengths;
```

Пример

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

//ново име на типът struct point - Point
typedef struct point Point;

typedef struct rect {
    struct point pt1; //възможни са и двата варианта
    Point pt2;
} Rectangle; //ново име на типът struct rect - Rectangle
```

Използваме новите имена:

```
Rectangle makeRectangle (Point pt1, Point pt2);
```

*typedef*₂

- **typedef** не създава нов тип - добавя ново име за вече съществуващ тип
 - **typedef** се използва, когато се пише код, който трябва да бъде преносим
 - Ако **typedef** се използва за типове данни зависещи от архитектурата, когато програмата се премести на нова система, единствено необходимо е да се сменят **typedef** декларациите
-
-

Обединения

- Обединението е променлива, която може да съдържа обекти от различен тип в различни моменти от време
 - Заделя се памет за най-голямото поле, различните променливи ползват една и съща (обща) памет
 - Общата памет се интерпретира по различен начин в зависимост от типа, с който се обръщаме към нея (ако е `int` – се интерпретира като `int`, ако е `float` – като `float` и т.н.)
-
-

Пример

```
union u_tag {
    int ival;
    float fval;
    char* sval;
} u;
...

#define INT 0 //ако пазим int
#define FLOAT 2 //ако пазим float
#define STRING 4 //ако пазим char *
int utype; //тук записваме какво в момента пазим в u
...
//някъде другаде (в utype) сме си записали в момента какви
//данни има в u и ги представяме в съответния формат
if (utype == INT)
    printf ("%d", u.ival);
if (utype == FLOAT)
    printf ("%f", u.fval);
if (utype == STRING)
    printf ("%s", u.sval);
```
