enum

enum е специален тип, чрез който се дават имена на група от конкретни константи. Константите се наричат *енумератори* и всяка от тях е от този специален тип. Могат имплицитно да се конвертират към `int` и са достъпни в целия scope, в който се намират.

Дефиниране и инициализация

Изброява се поредица от константи, като по подразбиране първата се приравнява на 0 и всяка е с 1 по-голяма от предходната.

```
enum Direction
{
    UP,   // = 0
    DOWN, // = 1
    LEFT, // = 2
    RIGHT // = 3
};
```

Може да се дадат конкретни стойности на някои/всички от константите

```
enum Color
{
    RED = -5,   // = -5
    GREEN,      // = -4
    BLUE,       // = -3
    YELLOW = 6, // = 6
    PURPLE,     // = 7
    BLACK       // = 8
}
```

Инициализират се както нормални променливи, *НО* стойността, която даваме може да е само една от изброените в групата!

```
// Валидни стойности
Direction direction = UP;
Color color = BLUE;

// Тези стойности не са валидни!
Direction invalid_direction = SOUTH;
Color invalid_color = ORANGE;
```

Struct

Можем да групираме произволни типове данни, които описват някаква обща концепция. Например една точка в двумерното пространство има координати (x, y). Удобно е да можем да разглеждаме конкретена точка като съвкупност от нейните координати.

Чрез структурите дефинираме тип, съставен от други типове. Обект от този тип съдържа в себе си обекти от другите типове, като те са последвателно наредени в паметта.

Дефиниране и инициализация

```
// Дефинираме нов тип "Point", който е съставен от "double" и "double".

struct Point
{
    double x;
    double y;
};
```

Типът може да се дефинира глобално и да се използва във всички scope-ове на програмата или да се дефинира локално и да се използва само в съответния scope.

Анонимни структури

```
// Обект "p" от тип анонимна структура, съставена от два "double"-а.

struct
{
    double x;
    double y;
} p;
```

Динамично заделени обекти и масиви

```
int main()
{
    Point* p = new Point;
    /*TODO*/
    delete p;

    Point* arr = new Point[10];
    /*TODO*/
    delete[] arr;

    return 0;
}
```

Членове

В дадените по-горе примери членовете са `x` и `y`. - Достъпваме елементите за *конкретен* обект `Point p` чрез оператор `'.'` така: **`p.x`**, **`p.y`**. - Ако имаме указател **`Point* p_ptr`**, достъпваме елементите на обекта, към който сочи указателя чрез оператор `'->'` така: **`p->x`**, **`p->y`**.

Членовете на константни обекти не могат да бъдат променяни.

Фунцкии

Структурите, както простите типове, могат да се подават на функции. Обектите могат да са много обемни, за това не искаме при всяко подаване да се копират.

- Подаваме чрез псевдоним

```
void setPoint(Point& p)
{
    std::cin >> p.x;
    std::cin >> p.y;
}

void printPoint(const Point& p)
{
    // тук не можем да променяме стойностите на 'x' и 'y' за обекта 'p'
    std::cout << '(' << p.x << ", " << p.y << ')' << std::endl;
}

int main()
{
    Point p;
    setPoint(p);
    printPoint(p);

    return 0;
}
```

- Подаваме чрез указател

```
void setPoint(Point* p)
{
    std::cin >> p->x;
    std::cin >> p->y;
}

void printPoint(const Point* p)
{
    std::cout << '(' << p->x << ", " << p->y << ')' << std::endl;
}

int main()
{
    Point p;
    setPoint(p);
    printPoint(p);

    return 0;
}
```

Динамично заделени членове

Ако ни се наложи да имаме динамично заделен масив в обекта, трябва да се грижим за паметта!

```
const size_t MAX_SIZE = 30;

struct Person
{
    char* first_name_;
    char* last_name_;
    size_t age;
};

void alloc_person(Person& p)
{
    p.first_name_ = new char[MAX_SIZE];
    p.last_name_ = new char[MAX_SIZE];
}

void dealloc_person(Person& p)
{
    delete[] p.first_name_;
    delete[] p.last_name_;
}
```