# Структуры

**Структура (struct)** — это композитный тип данных, представляющий собой набор из независимых переменных. Переменные, из которых составлена структура, называются её **полями** или **элементами** Каждое поле имеет свой тип и имя. Элементом структуры может быть переменная какого угодно типа, кроме самой структуры.

## Объявление

Объявляется структура следующим образом:

```
struct /*имя структуры*/
{
	/*поля структуры*/
} /*объявляемые переменные данного типа*/;
```

Имя структуры является необязательным параметром, но если его написать, то в дальнейшем эта структура будет существовать как самостоятельный тип данных, и переменные можно будет объявлять следующим образом:

```
struct /*имя структуры*/ /*объявляемые переменные*/;
```

Использование же безымянных структур является крайне редким явлением.

Сразу во время объявления структуры можно объявить и переменные, которые будут иметь такой тип. После переменных или, если их нет, после закрывающей фигурной скобки ставится точка с запятой.

Структуры лучше объявлять вне функций, поскольку это позволяет делать функции, возвращающие значение структурного типа.

## Обращение

После того, как были объявлены переменные структурного типа, к их полям надо обращаться. Для обращения к элементу структуры существует оператор **.** (точка). Чтобы обратиться к полю нужно написать `<имя переменной>.<имя поля>`. Следует понимать, что разные переменные одного и того же структурного типа имеют одинаковые имена соответствующих полей.

## Структуры и массивы

### Массивы в структурах

Элементом структуры может быть и массив. Причём, если это статический массив, размер которого определён одновременно с объявлением структуры, то весь массив будет содержаться непосредственно внутри каждой переменной данного типа. Если же это динамический массив, то структура хранит только указатель на него, сам массив может находиться где угодно.

В любом случае, обращение к элементу массива, являющегося элементом структуры, выглядит так: `<имя переменной>.<имя поля-массива>[<индекс>]`.

### Массивы структур

Из структур можно составить массив точно так же, как и из обычных переменных. При этом обращение к полю элемента массивы будет выглядеть так: `<имя массива>[<индекс>].<имя поля>`.

## Пример

У нас есть структура, в который хранятся точки на плоскости: координаты x и y и информация о цвете — яркость красной, синей и зелёной составляющих:

```
struct point
{
	int x, y;
	float r, g, b;
};
```

Объявлен массив shape длины 50 из этих точек:

```
struct point shape[50];
```

Задача: вывести этот массив на экран.

Графическая библиотека OpenGL содержит функцию `glVertex2i(int x, int y)`, которая в нашем случае будет означать вывод на экран точки с координатами x и y. Чтобы установить цвет выводимой точки, нужно перед вызовом этой функции вызвать другую функцию — `glColor3f(float red, float green, float blue)`, которая назначает цвет всех выводимых после неё точек вплоть до её повторного появления.

Опустив процедуру инициализации графического движка, получим такое решение задачи вывода массива на экран:

```
glBegin(GL_POINTS); //сообщаем графическому движку, что собираемся выводить точки
```

```
for(i=0; i<50; i++)
{
	glColor3f(shape[i].r, shape[i].g, share[i].b); //назначаем текущей точке цвет
	glVertex2i(share[i].x, shape[i].y); //выводим точку на экран
}
glEnd(); //сообщаем графическому движку, что рисование точек закончено
```

## Структуры и указатели

Элементом структуры не может быть сама эта структура, но может быть указатель на неё. связано это с тем, что структура, содержащая сама себя — это объект бесконечного размера, а вот размер указателя для каждой архитектуры всегда одинаков, поэтому размер структуры не не зависит от того, находится там указатель на int или на эту структуру.

Например, правильной будет такая структура:

```
struct node
{
	char* data;
	struct node* next;
};
```

Представим теперь, что нужно обратиться к символу, лежащему по адресу, указанному в какой-то переменной типа struct node, на которую ссылается переменная a этого же типа. Подобное обращение выглядит следующим образом:

```
*((*(a.next)).data);
```

Это, конечно, неудобно. Поэтому существует оператор `->` (знак минуса и закрывающей треугольной скобки), одновременно выполняющий операцию разадресации и обращения к элементу структуры. С его использованием это же обращение выглядит так:

```
a->next->data;
```

Гораздо короче и понятнее. Можно делать и такие обращения:

```
a->next->next->next->data;
```

Без использования оператора `->` оно выглядело бы так:

```
*((*((*((*(a.next)).next)).next)).data);
```

Данный пример делает очевидными преимущества применения оператора `->`.

## Определение новых типов данных

Иногда для удобства или повышения логичности программы требуется, чтобы какой-то из типов данных назывался по-другому. Это можно реализовать с помощью директивы define, но есть средство лучше.

Оператор `typedef тип1 тип2;` делает тип2 синонимом типа1. Тип1 для этого должен быть определён ранее. Например:

```
typedef unsigned char bool;
```

Такой код объявит новый тип данных bool, являющийся синонимом unsigned char.

typedef позволяет удобнее работать со структурами. Например, такое объявление:

```
typedef struct NODE
{
	char* data;
	struct NODE* next;
} node;
```

После него вместо struct NODE можно будет писать просто node, что короче и удобнее. Надо отметить, что внутри структуры тип node ещё не существует, поэтому там в любом случае надо писать struct NODE.

## Объединения

В языке Си существует ещё один композитный тип данных – объединение. Объединение отличается от структуры тем, что все его элементы находятся по одному адресу. При этом тип они могут иметь разный. То есть, объединение позволяет обращаться к одному и тому же набору байт памяти, интерпретируя его различным образом.

Размер объединения равен размеру его самого большого элемента.

Синтаксис объединений такой же, как и структур, но вместо «struct» пишется «union».

### Объединения и функции

Объединения позволяют, в частности, делать функции, возвращающие различные значения. Например, у нас есть строка, в которой содержится число. Но мы не знаем – целое оно или с плавающей точкой. Нужна функция, которая будет это определять и возвращать значение числа.

Для начала, создаём такую структуру:

```
typedef struct
{
	char is_float;
	union
	{
		float f;
		int i;
	} value;
} number;
```

В зависимости от значения поля is_float, из поля value будет считываться значение f или i.

Функция получится такая:

```
number aton(char* string)
{
	number result;
	int i;
	for(i=0; string[i]; i++)
	{
		if(string[i]=='.')
		{
			result.is_float=1;
			result.value.f=atof(string);
			return result;
		}
	}
	result.is_float=0;
	result.value.i=atoi(string);
	return result;
}
```

Поскольку строка содержит число по условию, то признаком того, что это число с плавающей точкой будет появление в строке символа точки. В случае её появления is_float устанавливается равным 1, а значение записывается как float. Если же в последовательнсоти символов точка не встретилась, то функция работает с ней как с целым числом.

# Работа с двоичными данными

## Битовые поля

Язык Си позволяет сделать элементом структуры переменную меньше одного байта. Для того чтобы это сделать надо написать так: `<целочисленный беззнаковый тип данных> <имя поля> : <ширина поля в битах>`.Такое поле будет называть **битовым полем**. Вне структуры обращение к нему будет идти как к указанному типу, но внутри неё оно будет иметь указанную ширину. Делать битовые поля имеет смысл только тогда, когда их несколько: тогда они будут находиться внутри одного байта, если же битовое поле одно, то под него всё равно будет выделено целое число байт.

Например, у нас есть такая структура:

```
typedef struct
{
	unsigned char e : 1; //равно
	unsigned char ne : 1; //не равно
	unsigned char g : 1; //больше
	unsigned char ge : 1; //больше или равно
	unsigned char l : 1; //меньше
	unsigned char le : 1; //меньше или равно
} flags;
```

Она характеризует отношение двух чисел. Каждое из полей может быть равным только 0 или 1, поэтому выделять под него целый бай неразумно. Тогда можно написать функцию, сравнивающую числа:

```
flags cmp(int n1, int n2)
{
	flags result;
	result.e=0;
	result.ne=0;
	result.g=0;
	result.ge=0;
	result.l=0;
	result.le=0;
	if(n1==n2)
	{
		result.e=1;
		result.ge=1;
		result.le=1;
		return result;
	}
	result.ne=1;
	if(n1<n2)
	{
		result.l=1;
		result.le=1;
		return result;
```

```
        }
        result.g=1;
        result.ge=1;
        return result;
}
```

В начале функции все битовые поля заполняются нулями по отдельности. Это не рационально. Можно создать объединение из структуры flags и типа char. Присвоив переменной типа char значение 0, мы автоматически обнулим все битовые поля:

```
flags cmp(int n1, int n2)
{
        union
        {
                flags f;
                char c;
        } result;
        result.c=0;
        if(n1==n2)
        {
                result.f.e=1;
                result.f.ge=1;
                result.f.le=1;
                return result.f;
        }
        result.f.ne=1;
        if(n1<n2)
        {
                result.f.l=1;
                result.f.le=1;
                return result.f;
        }
        result.f.g=1;
        result.f.ge=1;
        return result.f;
}
```

## Битовые маски

Другой путь работы с двоичными данными – это битовые маски.

Рассмотрим пример: нам надо поменять регистр всех букв в строке. «Лобовой» пусть решения – писать конструкции такого вида:

```
if(string[i]='a') {string[i]='A'; continue;}
if(string[i]='A') {string[i]='a'; continue;}
```

И так для каждой буквы. Решения подобного вида называются **брутфорс-решениями** (от англ. brute force – грубая/тупая сила), обладают крайне низкой эффективностью (в данном случае: если встретится буква z, то придётся перебирать весь латинский алфавит), и их применение является крайне нежелательным.

Если же рассмотреть таблицу символов ASCII, то можно заметить, что все символы верхнего регистра лежат в диапазоне от 0x41 до 0x5A, а нижнего – от 0x61 до 0x7A и расположены они в одинаковом (алфавитном) порядке. Это значит, что младшие тетрады у символов верхнего и нижнего регистра одинаковые, отличаются только старшие.

Верхний регистр $4_{16}=0100_2$ $5_{16}=0101_2$
Нижний регистр $6_{16}=0110_2$ $7_{16}=0111_2$

Видно, что у символов верхнего регистра второй бит старшей тетрады равен 0, а у нижнего – 1. Поэтому достаточно изменить его значение на противоположное, чтобы изменить регистр.

Вспомним теперь свойства операции ИСКЛЮЧАЮЩЕЕ ИЛИ: x^1=~x, x^0=x. То есть, чтобы изменить значение одного бита числа на противоположное, нужно применить к нему операцию побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ с числом, у которого все биты равны 0, кроме того, который нужно изменить. Иными словами, достаточно ко всем элементам строки применить такую операцию:

```
string[i]^=0x20;
```