# Семинар #4: Часть 2: Структуры. Классные задачи.

## Основы структур

Структуры – это композитный тип данных, объединяющий набор объектов в один объект. Используя структуры, мы можем сами создать новый тип данных, используя другие типы как кирпичики. Предположим, что мы разрабатываем приложение для работы с двумерной графикой и нам понадобилось как-то описывать точку в двумерном пространстве. Для этого мы можем опеределить структуру точки, как это сделано в следующем примере:

```c
#include <stdio.h>

struct point
{
    float x;
    float y;
};

int main()
{
    struct point a = {0.5, 1.5};
    a.x = 2.0;
    printf("(%g, %g)\n", a.x, a.y); // Напечатает (2.0, 1.5)
}
```

**Пояснение по коду:**

- Сначала мы определили новую структуру с двумя полями `x` и `y` типа `float`. После того, как мы это сделали у нас появился новый тип данных по имени `struct point` (да, название типа состоит из двух слов).
- **!** Не забывайте ставить точку с запятой в конце определения структуры. Это нужно делать обязательно.
- Далее, в функции `main`, создаём переменную типа `struct point` по имени `a` и сразу инициализируем её.
- Переменная `a` типа `struct point` хранит в себе два объекта типа `float` по имени `x` и `y`.
- Получить доступ к внутренностям переменной `a` можно с помощью оператора `.` (точка).

## Допустимые операции со структурами

1. При создании структуры её элементы можно инициализировать с помощью фигурных скобочек.

   ```c
   struct point a = {1.0, 2.5};
   ```

   Однако нельзя таким образом присваивать

   ```c
   a = {2.0, 1.0}; // Ошибка, фигурными скобками можно только инициализировать!
   ```

2. Доступ к элементу структуры осуществляется с помощью оператора точка

   ```c
   a.x = 5.5;
   a.y = 3.0;
   ```

3. Структуры можно присваивать друг другу. При этом происходит побайтовое копирование содержимого одной структуры в другую.

   ```c
   struct point b;
   b = a;
   ```

## Массив структур

Структуры, как и обычные переменные, можно хранить в массивах. В примере ниже создан массив под названием `array`, содержащий в себе 3 точки.

```c
#include <stdio.h>

struct point
{
    float x
    float y;
};

int main()
{
    struct point array[3] = {{1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6}};
    array[1].y = 9.9;
    printf("(%g, %g)\n", array[1].x, array[1].y); // Напечатает (3.3, 9.9)
}
```

## Передача структуры в функцию

Структуры можно передавать в функции и возвращать из функций также как и обычные переменных. При передаче в функцию происходит полное копирование структуры и функция работает уже с копией структуры. При возвращении из функции также происходит копирование.

```c
#include <stdio.h>

struct point
{
    float x;
    float y;
};

void print_point(struct point a)
{
    printf("(%g, %g)", a.x, a.y);
}

struct point add_points(struct point a, struct point b)
{
    struct point result;
    result.x = a.x + b.x;
    result.y = a.y + b.y;
    return result;
}

int main()
{
    struct point a = {1.1, 2.2};
    struct point b = {3.3, 4.4};
    struct point c = add_points(a, b);
    print_point(c);
}
```

### Структуры содержащие более сложные типы данных

Структуры могут содержать в себе не только базовые типы данных, но и более сложные типы, такие как массивы (в том числе строки), указатели, а также другие структуры.
Пример программы, в которой описывается структура для удобной работы с объектами Книга (`struct book`).

```c
#include <stdio.h>
#include <string.h>

struct book
{
    char title[50];
    int pages;
    float price;
};

void print_book(struct book b)
{
    printf("Book info:\n");
    printf("Title: %s\nPages: %d\nPrice: %g\n\n", b.title, b.pages, b.price);
}

int main()
{
    struct book a = {"The Martian", 10, 550.0};
    print_book(a);

    a.pages = 369;
    strcpy(a.title, "The Catcher in the Rye");
    print_book(a);

    struct book scifi_books[10] =
    {
        {"Dune", 300, 500.0},
        {"Fahrenheit 451", 400, 700.0},
        {"Day of the Triffids", 304, 450.0}
    };
    scifi_books[2].price = 2000.0;
    print_book(scifi_books[2]);
}
```

### Создаём более удобное имя для типа структуры, используя `typedef`

По умолчанию для структуры создаётся имя типа, состоящее из двух слов, например `struct book`. Это может быть не очень удобно, так как использование такого имени делает ваш код многословным. Чтобы укоротить имя типа можно использовать ключевое слово `typedef`:

```c
struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;
// После этого можно использовать имя Book для названия типа структуры
```

## Указатели на структуры:

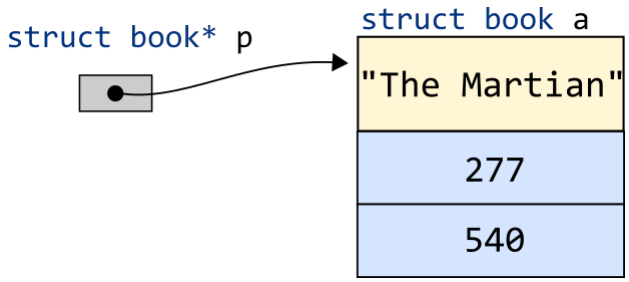
Указатель на структуру хранит адрес первого байта структуры. Для доступа к полям структуры по указателю нужно сначала этот указатель разыменовать, а потом использовать: `(*p).price`. Для удобства был введён оператор стрелочка `->`, который делает то же самое: `p->price`.

```c
#include <stdio.h>

struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;

int main()
{
    Book a = {"The Martian",277,540};
    Book* p = &a;

    // Три способа доступа к полю:
    a.price += 10;
    (*p).price += 10;
    p->price += 10;
}
```



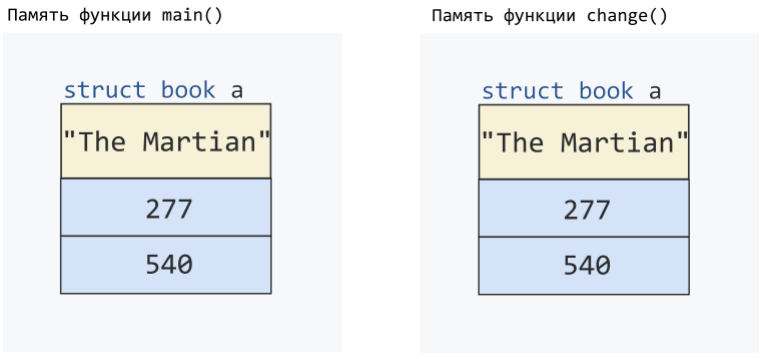

### Передача по значению

При обычной передаче в функцию всё содержимое копируется. Функция работает с копией.

```c
#include <stdio.h>

struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;

void change(Book a)
{
    a.price += 10;
}
int main()
{
    Book a = {"The Martian",277,540};
    change(a); // a НЕ изменится
}
```

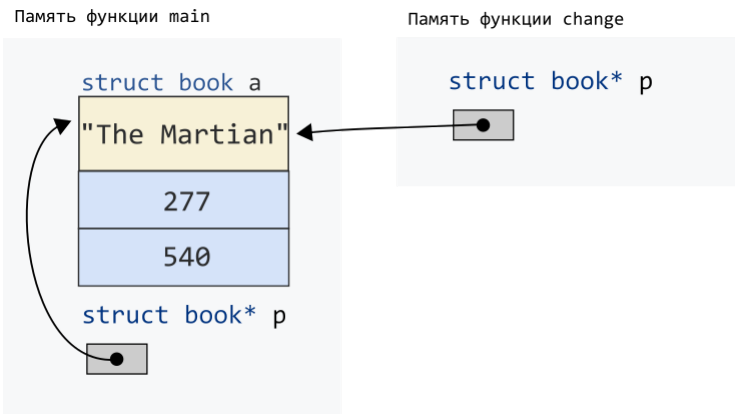


## Передача по указателю

При передаче в функцию по указателю копируется только указатель.

```c
#include <stdio.h>
struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;

void change(Book* p)
{
    p->price += 10;
}
int main()
{
    Book a = {"The Martian",277,540};
    Book* p = &a;
    change(p); // внутри функции структура a
    изменится
}
```




Такой способ передачи имеет 2 преимущества:

1. Можно менять структуру внутри функции, и изменения будут действительны вне функции
2. Не приходится копировать структуры, поэтому программа работает быстрее.

## Передача по константному указателю

Иногда мы не хотим менять структуру внутри функции, но хотим чтобы ничего не копировалось. Тогда желательно использовать передачу по константному указателю.

```c
#include <stdio.h>
struct book
{
    char title[50];
    int pages;
    float price;
};
typedef struct book Book;

void print_book_info(const Book* p)
{
    printf("Title: %s\nPages: %d\nPrice: %g\n\n", p->title, p->pages, p->price);
}

int main()
{
    Book a = {"The Martian",277,540};
    print_book_info(&a);
}
```

# Выравнивание

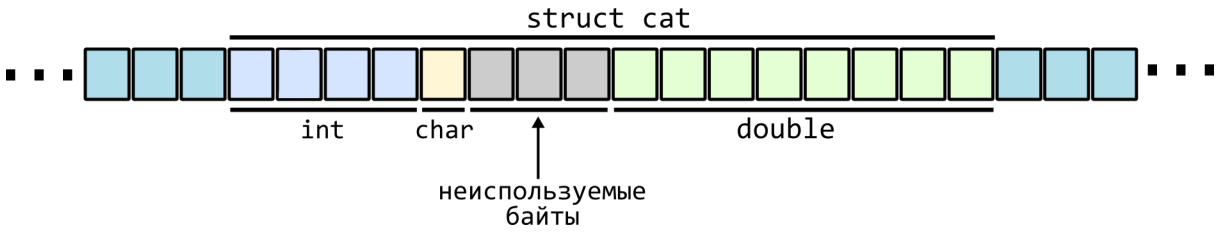
Пусть есть структура `struct cat` и нам нужно узнать её размер, если размеры типов `char`, `int` и `double` равны `1`, `4` и `8` байт соответственно.

```
struct cat
{
    int a;
    char b;
    double c;
};
```

Кажется, что размер этой структуры равен сумме размеров состовляющих её элементов, то есть 13 байт, но это не так. На самом деле, размер этой структуры будет отличаться в зависимости от вычислительной системы, на которой запускается код (как, впрочем, и размеры других типов). Но на большинстве вычислительных систем размер структуры `cat` будет равен 16 байт. Это можно проверить с помощью следующего кода:

```
#include <stdio.h>
struct cat
{
    int a;
    char b;
    double c;
};
int main()
{
    printf("Size of char   = %zu\n", sizeof(char));
    printf("Size of int    = %zu\n", sizeof(int));
    printf("Size of double = %zu\n", sizeof(double));
    printf("Size of cat    = %zu\n", sizeof(struct cat));
}
```

Так происходит потому что компьютер работает более эффективно с объектами, которые лежат в памяти по адресу, кратному некоторой величине, называемой выравниванием. Например, с числами типа `double` компьютер работает более эффективно если они лежат по адресам, кратным 8-ми. Поэтому в памяти структура `cat` выглядит так:




Можно считать, что каждый тип данных, помимо размера, характеризуется ещё одной величиной - выравниванием. Выравнивание - это некоторое значение в байтах. Оно означает, что объекты данного типа будут располагаться в памяти по адресам, кратным выравниванию.

Для того, чтобы найти величину выравнивания можно использовать оператор `alignof` или `_Alignof`:

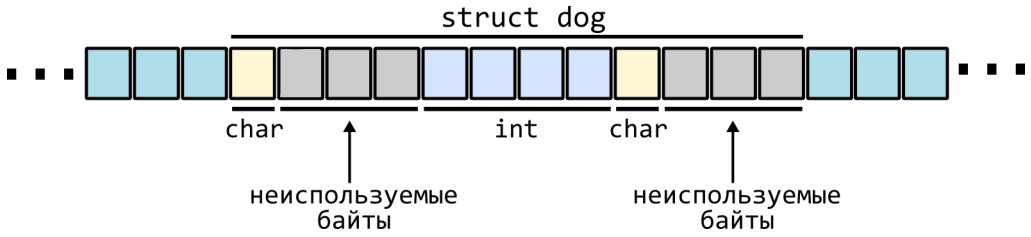
```
#include <stdio.h>

int main()
{
    int a = 10;
    printf("Alignment of int = %zu\n", _Alignof(int));
    printf("Alignment of int = %zu\n", _Alignof(a));
}
```

Рассмотрим ещё пример выравнивания для следующей структуры:

```
struct dog
{
    char a;
    int b;
    char c;
};
```

Эта структура на большинстве систем будет занимать 12 байт и в памяти будет выглядеть вот так:




Выравнивание у типа `char` равно 1, поэтому объекты этого типа могут располагаться где угодно, а выравнивание у типа `int` равно 4, поэтому объекты этого типа желательно располагать по адресам, кратным четырём. Это объясняет, почему при расположении числа типа `int` после числа типа `char` был сделан отступ в 3 байта.

Также обратите внимание, что в этом случае неиспользуемые байты были добавлены в конец структуры. Зачем это было нужно? Представьте, что мы создали массив из структур `dog`. Все элементы массива в памяти должны лежать плотно примыкая друг к другу, при этом все поля всех структур в массиве должны быть выравнены. Это можно добиться только добавив три неиспользуемых байта в конец структуры.

Интересно, что размер структуры может зависеть от порядка полей структуры. Например, если мы просто поменяем порядок полей в структуре `dog` вот так:

```
struct dog2
{
    char a;
    char c;
    int b;
};
```

то размер этой структуры уже будет равен 8 байт, а в памяти структура будет выглядеть следующим образом: