Лекция 04 — Структуры и объединения

”Вчера отчислили ваших коллег!”
— А. П. Хвастунов

Структуры

> **Info**
>
> **Структура** — одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем.

Синтаксис объявления и определения структур:

```
struct Point; // объявление

struct Point{  // определение
    int x; // переменные, которые содержатся в структуре
    int y;
} pt1, pt2, pt3; // объявляются глобальные переменные pt1, pt2, pt3 типа
Point (указывать экземпляры сразу после объявления не обязательно, но точка
с запятой в конце нужна всегда)

Point p4;
```

> **Info**
>
> Объявленные в структуре переменные называются её **полями**.
>
> Переменные, принимающие тип данных, заданный структурой, называются её **экземплярами**.

Обращение к полям экземпляра структуры реализуется при помощи оператора `.`:

```
Point pt;
pt.x = 200;
pt.y = 300;
```

Вложенные структуры

Мы можем использовать в качестве полей структуры другие структуры или даже объявить одну структуру внутри другой. При этом все взаимодействия со вложенными структурами происходят ровно так же, как и с обычными:

```
struct Rectangle {
    Point p1;
    Point p2;

    struct Diameter {
        int length;
    } d;
}

int main() {
    Rectangle rec;
    Point pt;
    rec.pt1 = pt;
    rec.pt1.x = 2;
    rec.d.length = 238;
}
```

Важно понимать, что даже если мы объявили какую-то вложенную именованную структуру, то мы сможем обратиться к её полям лишь тогда, когда будет создан хотя бы один её экземпляр.

Анонимные структуры

Если нам по какой-то причине оказалось необходимо разбить поля структуры на какие-то логические блоки и при этом мы не хотим прибегать к использованию именованных вложенных структур, то их можно сделать *неименованными* (или *анонимными*):

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

    struct {
        size_t width;
        size_t height;
    };
};
```

```
Button btn = {.x = 0, .y = 100, .width = 400, .height = 80};
```

Если и только если мы не объявили ни одного экземпляра неименованной структуры, мы можем обращаться к её полям напрямую, как если бы это было поле внешней структуры.

Инициализация структуры

Существует два способа инициализировать структуру: воспользоваться *списком инициализации*[1] или явно обращаться к полям в фигурных скобках. Оператор присваивания при этом можно опускать:

```
Point p1 = {1, 2};
Point p2 {1, 2};
Point p3 = {.x = 1, .y = 2}; // порядок не важен
Point p4 {.x = 1, .y = 2};
// можно мешать 1 и 3 способ, но лучше так НЕ делать

Rectangle r1 = {{1, 2}, p2};
Rectangle r2 = {.pt1 = {1, 2}, .pt2 = p2};
```

Массивы структур

Мы можем объявлять массивы данных заданного структурой типа:

```
struct Record {
    char name[10];
    char surname[10];
    long phone;
}

Record phonebook[200];
```

Указатели на структуры

Как и с любым другим объектом, мы можем получить адрес структуры и задать соответствующий указатель:

```
Record* FindRecord(long phone, Record* records, int count) {
    for (int i = 0; i < count; ++i) {
        if (records[i].phone == phone) return &records[i];
    }
```

```
    return nullptr;
}
```

Если мы хотим обратиться к полю структуры по указателю на экземпляр, то вместо оператора `.` необходимо использовать оператор `->`:

```
struct Foo {
    int x;
};

int main() {
    Foo foo;
    foo.x = 237;
    std::cout << foo.x << '\n';
    Foo* foo_pointer = &foo;
    std::cout << foo_pointer->x << '\n';
    foo_pointer->x += 1;
    std::cout << foo.x << '\n';
}
```

Выравнивание структур

На первый взгляд может показаться, что поля структуры будут храниться в памяти последовательно, одно за другим. На деле с большой вероятностью окажется так, что структура будет занимать значительно больше места, чем, казалось бы, должна была, если бы мы исходили из такого принципа:

```
struct Foo1 {
    char a; // 1 byte
    int64_t b; // 8 bytes
    int8_t c; // 1 byte
    int32_t d; // 4 bytes
};

// sizeof(Foo1) == 24
```

Более того, мы можем обнаружить, что размер структуры зависит от порядка объявления её полей:

```
struct Foo2 {
    int64_t b; // 8 bytes
    int32_t d; // 4 bytes
    char a; // 1 byte
```

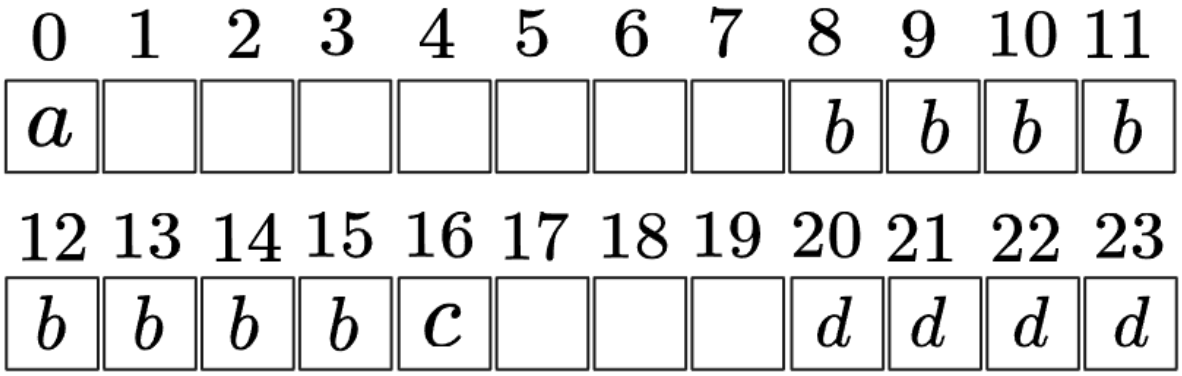
```
    int8_t c; // 1 byte
};

// sizeof(Foo2) == 16
```

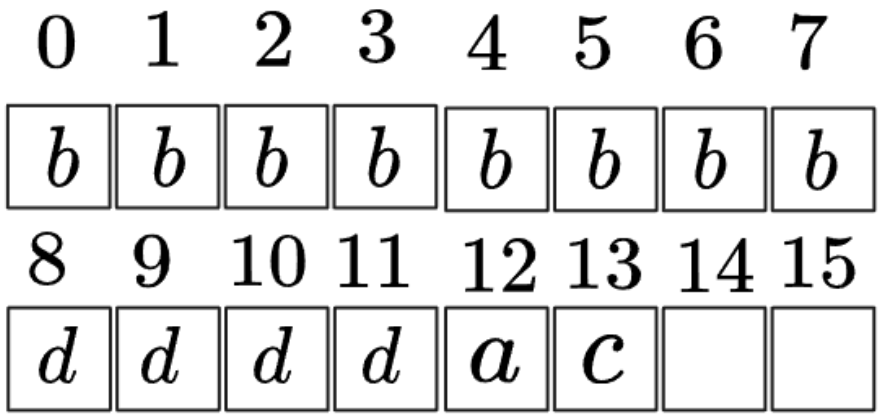
Дело в том, что из соображений производительности компилятор может вставлять между полями и в конце структуры пустые байты — это называется **structure padding**. Механизм следующий:

1. Каждое поле выравнивается по адресу, кратному размеру самого поля.
2. Общий размер структуры округляется вверх до кратного размеру самого крупного поля, чтобы массивы таких структур были также корректно выровнены.

Так, структура `Foo1` будет выглядеть в памяти следующим образом:



В то же время, структура `Foo2` при выравнивании по всем тем же правилам будет выглядеть так:



> ⚠ **Warning**
>
> Из механизма выравнивания следует, что поля структур наиболее оптимально объявлять *по невозрастанию их размера*.

Если мы по какой-то причине хотим повлиять на выравнивание, то можно воспользоваться следующими инструментами:

- для уменьшения выравнивания — директива препроцессора `#pragma pack(c)`, где `c` — константная величина, равная 1, 2, 4, 8 или 16. При её использовании выравнивание будет происходить так, чтобы адреса переменных были кратны `c` — соответственно, указание директивы `#pragma pack(1)` отключает паддинг. Прервать её действие можно при помощи вызова директивы `#pragma pack()` без аргументов, которая установит выравнивание по умолчанию.
  Мы также можем использовать директивы `#pragma pack(push, c)` и `#pragma pack(pop)`, которые реализуют всё описанное выше на стеке, позволяя откатываться к предыдущим настройкам выравнивания.

```
#pragma pack(1)

struct Foo1 {
  char a;
  int64_t b;
  int8_t c;
  int32_t d;
}

// sizeof(Foo1) == 14

#pragma pack()

struct Foo2 {
  char a;
  int64_t b;
  int8_t c;
  int32_t d;
}

// sizeof(Foo2) == 24
```

- для увеличения выравнивания — спецификатор `alignas(c)`, где `c` — константная степень двойки, причем `c` больше, чем размер выравнивания, предусмотренный компилятором.

```
struct alignas(64) Foo {
  char a;
  int64_t b;
  uint8_t c;
  uint32_t d;
}
```

```
// sizeof(Foo) = 64
```

Важно: `alignas` перекрывает действие `#pragma pack()`.
Ещё более важно: на практике мы вообще не хотим вмешиваться в процесс выравнивания, потому что это почти гарантированно приведет к потере производительности и, возможно, даже к ошибкам при обращении к памяти. Воспринимайте этот блок как бесполезную статью в научно-популярном журнале.

Объединения

Объединение (*union*) — это переменная, позволяющая хранить разнородные данные в одной и той же области памяти, то есть способная в разные моменты времени содержать в себе объекты различных типов и размеров. Все требования относительно размеров и выравнивания при этом выполняет компилятор.

```
union Name {
    struct {
        char name[13];
        char code[3];
    }

    struct {
        int32_t i1;
        int32_t i2;
        int32_t i3;
        int32_t i4;
    }
};
```

Из определения несложно заметить, что размер юниона будет с точностью до выравнивания равен максимуму из размеров его полей:

```
union U {
    int a;
    double b;
    char c[13];
};

// sizeof(U) == 16
```

В ряде моментов с юнионами можно работать как со структурами. В частности, мы можем:

- обращаться к их полям[2];
- передавать их в функции;
- делать их анонимными и др.

Более подробно с концепцией юниона и тонкостями работы с ним можно ознакомиться в документации. [3]

---

1. https://en.cppreference.com/w/cpp/language/list_initialization.html ↩
2. Строго говоря, обращение к любому члену юниона, кроме активного (т.е. последнего записанного), не регулируется стандартом и является UB (за исключением случая записи в неактивный член). Если мы хотим одновременно обращаться к нескольким полям, т.е. интерпретировать один участок памяти как несколько разных типов, то мы должны сами гарантировать, что эта операция будет корректной. ↩
3. https://en.cppreference.com/w/cpp/language/union.html ↩