1 Алгебраические типы данных

Это не слишком важно, но стоит знать, чтобы понять `struct, union, enum`.
У нас есть: тип-сумма и тип-произведение. Без формальных определений, примеры:

тип-сумма: Такой тип, количество возможных значений которого определяются суммой количеств вохможных значений его составляющих. Например: `int`. Если мы скажем, что каждое целое число - это СВОй тип (что иногда делается в математике), а количество значений типа n - 1 то `int` состоит из типов $0, 1, 2, \ldots, 2^{32}$, и имеет ровно 2^{32} значений

тип-произведеине: такой тип, количество возможных значений которого есть произведение количества значений типов, из которых он состоит. Например, Вектор в двумерном пространстве. Каждый Вектор состоит из 3 чисел - координат - то есть `vector = (int, int, int)`. Как видно, количество возможных значений такого вектора = $(2^{32})^3 = 2^{96}$

Пу сути, это всё, что нужно знать про это тему. А теперь к самому соку:

2 Enum

Энумерация - это *тип-сумма*, каждому элементу которого мы даём название. Объявление:

```
enum time_of_day {
        night, morning, afternoon, evening
}; // Не забываем запяточку, товарищи)
```

Здесь мы говорим, что тип `time_of_day` может состоять только из ОДНОГО из 4 значений: ночь, утро, день, вечер. Внутри, компьютер присваивает каждому из 4 значений целочисленное значение. По крайней мере у меня на компе, они начинают с нуля и идут с шагом по единце, то есть `night=0, morning=1, afternoon=2, evening=3`. Мы можем это изменить:

```
enum time_of_day {
        night = 1, morning, afternoon = 10, evening = 15
};
int main() {
        enum time_of_day current_time = night;
        return 0;
}
```

Здесь `night=1, morning=2, afternoon=10, evening=15`. Заметьте, если мы не указали значение, то оно устанавливается на предыдущее+1 (если это не вызовет повторов)
Переменная `current_time` сейчас содержит целое число `1`. Перед объявлением переменной с типом `time_of_day` **надо** писать слово `enum`. Если у вас работает и без этого слова, Пояснение в конце документа. Как сделать так, чтобы работало без слова, там же.

А теперь вопрос, нахуя? Есть 2 основных использования:

2.1 Enum Флаги

```
enum file_permission {
        FP_READ  = 1 << 1, // 1
        FP_WRITE = 1 << 2, // 2
        FP_EXEC  = 1 << 3, // 4
        // Если бы было больше полей, они были бы 8, 16, 32, 64...
        // то есть степени двойки
};
void create_file(const char *path, int permissions); // допустим, она где-то есть
int main() {
        // создаём файл, который можно читать и в который можно писать
        create_file("C:\\Windows\\System32\\rickroll.png", FP_READ | FP_WRITE);
}
```

Поскольку мы сказали, что каждое значение в `enum` - это 2^n, то в двоичной записи при **побитовом сложении** (если не знаете, что это, почитайте), каждый n-ый бит отвечает за конкретное значение `file_permission`. Чтобы проверить, какие значения были установлены, обычно делают так:

```
if (permissions & FP_READ) {...}
if (permissions & FP_WRITE {...}
...
```

То есть побитовым умножением (непонятно, почему? напишите на листочке, как это будет выглядеть для компа, и учитывайте, что любое число $\neq 0$ для компа - это `true`, и только 0 - это `false`)

Для задания 7 не особо нужно

2.2 Для switch

Сразу пример:

```
enum animal_type { AT_DOG, AT_CAT, AT_FJORD };
...
switch (get_animal_type(some_animal)) {
        case AT_DOG: puts("hey, I got a dog");
        case AT_CAT: puts("hey, I got a cat");
        case AT_FJORD: puts("what the fuck did i get");
}
```

То есть, это буквально набор констант. аналогично можно было бы сказать:

```
#define AT_DOG 0
#define AT_CAT 1
#define AT_FJORD 2
```

Но, согласитесь, пусть комп сам разбирается, какие значения ему куда приСВОить

Важное замечание!!! Когда вы объявляете члены `enum`, они становятся Глобальными Константами! То есть, если вы объявите член энумерации `black` и потом используете переменную с названием `black`, вы можете получить довольно неприятные и неожиданные результаты. Поэтому лучше всего ко всем именам добавлять префикс, а сами имена писать капсом.

3 Struct

Структура - это уже *тип-произведение*. Давайте воспользуемся примером из начальной секции:

```c
struct vector3 {
        float x;
        float y;
        float z;
}; /* запяточка!); */
void print_vector(struct vector3 vec) {
        printf("vector {%f; %f; %f}\n", vec.x, vec.y, vec.z);
}
int main() {
        struct vector3 myvec = {69, -1, 420};
        print_vector(myvec);
        return 0;
}
```

Что происходит? **В памяти компа** это хранится просто как 3 `float`-а подряд, поэтому структура занимает столько же места, сколько заняло бы 3 дробных числа. объединять связанные переменные в структуры помогает организовывать код и делать его более читабельным! Сейчас у нас есть вектор с координатами `{69, -1, 420}`. Попробуем написать что-то полезное:

```c
struct vector3 scalar_prod(struct vector3 a, struct vector3 b) {
        struct vector3 result = {
                .x = a.x * b.x,
                .y = a.y * b.y,
                .z = a.z * b.z
        };
        return result;
}
```

Обратите внимание на синтаксис: мы можем явно указать, в какое поле мы что записываем (`.field = value, ...`). Это не обязательно, но иногда полезно. Для доступа к полям `x, y, z` мы используем оператор точки.
Важно! Это одна из двух фич, которые есть в **C**, но нет в **C++**. Так что, если вы используете *Visual studio* и создаёте проект на C++, и ваш файл называется `main.cpp`, а не `main.c`, возможно, вы не сможете так делать. Обычно это решается тем, чтобы переименовать `main.cpp` в `main.c`.

Но что происходит, когда мы вызываем эту функцию? он копирует вектора `a` и `b`, и только потом передаёт их как аргументы. Поскольку часто структуры не такие маленькие, как наш `vector3`, то, чтобы избежать копирования, используем указатели:

```c
struct vector3 scalar_prod(const struct vector3 *a, const struct vector3 *b) {
        struct vector3 result = {
                .z = a->z * b->z
                .y = a->y * b->y,
                .x = a->x * b->x,
        };
        return result;
}
```

Чтобы вызвать эту функцию, пишем что-то вроде `scalar_prod(&vec1, &vec2)`

Первое: Когда вы передаёте что-то по указателю чтобы избежать копирования, лучше пишите `const`. Так, даже если вы **случайно** измените переменную a, компилятор вас предупредит и не скомпилирует (поверьте, вы *не хотите* проводить час на отладку программы из-за того, что опечатались и случайно написали `a->x = a->x * b->x`)

Второе: вместо точки мы пишем стрелочку. Когда мы работаем с указателями, все точки заменяются на стрелочки, это правило. можно было бы написать `(*a).z`, но это привело бы к копированию и выглядит некрасиво.

Третье: я решил поменять порядок, в котором записываю поля. Если мы не используем `{.field = value, ...}` и пишем просто `{value, ...}`, менять порядок инициализации мы не можем.

Практическое применение? думаю, `vector3` уже достаточный аргумент для полезности структур - вместо трёх переменных или массива, мы храним всю инфу в одной, с говорящими названиями. Они лежат в самом сердце языка и являются основной причиной, почему после **C** я просто не могу программировать на питоне. Мне их пиздец как не хватает там.

4 Union

Объединение - это *тип-сумма*. Это важное замечание, потому что многие часто не понимают, нахрена они вообще нужны. Они выглядят так же, как структуры, у них абсолютно аналогичный синтаксис, но, вместо того, чтобы хранить все поля, они хранят **Только одно из полей**. Пример:

```c
union float_or_int {
        float f;
        int i;
};
int main() {
        union float_or_int something = {.f = 10.0};
        printf("something.f = %f, something.i = %i\n", something.f, something.i);
        something.i = 10;
        printf("something.f = %f, something.i = %i\n", something.f, something.i);
        return 0;
}
```

Получаем вывод:

```
something.f = 10.000000, something.i = 1092616192
something.f = 0.000000, something.i = 10
```

То есть, что произошло? `union` хранит абсолютно все СВОи поля в *одной области памяти*. Это значит, что если мы пишем `something.f = 10.0`, то в пямяти хранится мантисса, экспонена и т.п., и когда мы читаем целое число из этой же области памяти, целое число получается бессмысленным.

Аналогично, когда записываем int, получаем, что float - это бессмыслица. Экспонента равна нулю, мантисса = 0x1010, поэтому число у нас около нуля. Получается, размер `union` - это размер самого большого поля. Вопрос, нафиг надо? ну, представьте ситуацию:

```c
int main() {
        union float_or_int array[] = {
                {.f = 10.0},
                {.i = -7},
                {.f = 1e48}, // 1 · 10^48
                {.i = 1000}
        };
}
```

То есть массив хранит в себе либо `float`, либо `int`. Но как понять, где что? давайте поступим так:

```
enum union_type {
        U_FLOAT, U_INT
};
int main() {
        union float_or_int array[] = {...};
        enum union array_types[] = {U_FLOAT, U_INT, U_FLOAT, U_INT};
}
```

Таким образом, когда проходимся по индексам `array`, мы по этим же индексам в массиве `array_types` можем понять, к чему нам обращаться - к полю `.f` или `.i`.

Но знаете, возникает вопрос - а можно ли это как-то объединить..? Итак, представляю вам невероятную вещь:

4.1 Tagged Union

Можно сказать, это самое полезное и, чуть ли не единственное применение `union` в языке **C**. В общем виде это структура, которая содержит в себе энумерацию и объединение. Энумерация позволяет определить, какой из членов объединения содержит валидные данные. Продолжая пример из прошлого раздела:

```
struct tagged_union {
        enum union_type type;
        union float_or_int data;
};
...
struct tagged_union vals[] = { {U_INT, .data.i = 10}, {U_FLOAT, .data.f = 10.0}, ..
for (int i = 0; i < sizeof vals / sizeof vals[0], ++i) {
        if (vals[i].type == U_INT)
                printf("%i\n", vals[i].data.i);
        else
                printf("%f\n", vals[i].data.f);
}
```

Примечание: они не спасают вас от возможности обосраться и перепутать типы! Вы вручную должны устанавливать правильный `type` и следить, что вы пишете в правильное поле (новые языки программирования вроде *Rust* делают это автоматически).

Итак, уже можно считать, что вы знаете минимум, чтобы сделать всё, что угодно. Но, чтобы жизнь была легче, надо потерпеть ещё немного. Итак,

5 Анонимные типы

Звучит страшно, но по сути это значит следующее: мы объявляем тип, но **не даём ему имени**. Как, зачем, почему? Допустим, вам нужно использовать какую-то структуру всего один раз и в одном месте. тогда можно написать

```
struct { int a, int b } my_variable;
my_variable.a = 10;
my_variable.b = 52;
```

То есть типа нет, но мы **сразу** объявляем переменную. Это можно сделать и глобально, и внутри функции, и, главное... Внутри другой структуры! И так можно сделать и с `union`, и с `enum`. ПользуемсяСВООими новыми знаниями, чтобы объявить *tagged union* по-новому

```
struct tagged_union {
        enum {TU_INT, TU_FLOAT} type;
        union {
                int i;
                float f;
        } data;
};
struct tagged_union un = {.type = TU_INT, .data.i = 0};
un.type = TU_FLOAT; un.data.f = 0.00001f;
```

Ну не красота ли? Поскольку ни в объединении, ни в энумерации нет смысла без этой структуры, иметь всё в одном месте и изолированно очень даже хорошо. Ладно. анонимные типы были классными, но настало время кое-чего ещё более крутого...

5.1 ”Анонимные” Анонимные типы

```
struct tagged_union {
        enum {TU_INT, TU_FLOAT} type;
        union {
                int i;
                float f;
        };
};
struct tagged_union un = {.type = TU_INT, .i = 0};
un.type = TU_FLOAT; un.f = 0.00001f;
```

Найдите 10 отличий. Мы видим, что у `union` внутри пропало даже имя переменной, и теперь `i` и `f` выглядят прямо как поля структуры - мы даже не подозреваем (не посмотрев на объявление структуры), что они на самом деле внутри `union`-а. Аналогично можно делать и со структурами, но это не работает с `enum`-ами.

По сути, правило: Если у внутреннего типа данных (`struct, union`) нет имени, то его поля становятся полями внешнего типа данных, но ведут себя и занимают место по ”СВОим” правилам.

Теперь детальный пример, который порадовал бы Ангелуца и пригодился бы в задании 7.2:

```
struct figure {
        enum {
                F_NONE, F_POINT, F_RECT, F_LINE, F_CIRCLE
        } type;
        union {
                struct {
                        vector2 coords;
                } point;
                struct {
                        vector2 upleft, downright;
                } rect;
                struct {
                        vector2 start, end;
                } line;
                struct {
                        vector2 center;
                        long rad;
                } circle;
        };
};
// draw для F_POINT
void draw_point(const struct figure *self, struct drawable *d);
// draw для F_RECT
void draw_rect(const struct figure *self, struct drawable *d);
// draw для F_LINE
void draw_line(const struct figure *self, struct drawable *d);
// draw для F_CIRCLE
void draw_circle(const struct figure *self, struct drawable *f);
// такой синтаксис - это расширение компилятора gcc, и не работает
// на винде. на винде вы бы просто писали так, чтобы массив по индексу
// F_POINT содержал функцию draw_point и т.п....
void (*const draw_lookup[])(const struct figure *, struct drawable *) = {
        [F_NONE] = NULL,
        [F_POINT] = draw_point,
        [F_RECT] = draw_rect,
        [F_LINE] = draw_line,
        [F_CIRCLE] = draw_circle
};
// чтобы не возвращать длину, последний элемент
// имеет тип F_NULL, и так мы понимаем, что массив закончился
// помните, как в заданиях ЕГЭ? читаем файл, пока не встретим 0
struct figure* read_file(FILE* f);

int main() {
        // это массив
        struct figure *fs = read_file(file);
        for (struct figure *i = fs; i->type != F_NONE; ++i)
                draw_lookup[i->type](i);
        free(fs);
}
```

Испугались? Я тоже... Функции не объявляю, потому что это выходит за рамки темы и полный код есть на моём gitbub.

Что делает код? 1. объявляем структуру, которая может хранить либо точку, либо прямоугольник, либо линию, либо круг. Далее *типа* создаём фунуции, которые будут принтить фигуру по её параметрам. Потом создаём массив, в котором мы храним эти функции **по индексу типа фигуры**. Аналогом в Питоне был бы словарь `(dict, {})`, где ключами были бы названия фигур, а значениями - названия функций. Только в **C** это очень оптимально.

6 Типы типов перед объявлением переменной

"Типом типа" я называю слова `enum, struct, union`. Итак, почему мы вообще пишем `struct vector3`, а не просто `vector3`? Разве и так не понятно?

На самом деле, понятно. И в **C++** писать тип типа *не нужно*. Так что, если у вас рабоатет не писать тип типа, значит, это потому, что вы используете компилятор **C++**, который является компилятором по умолчанию в *Visual Studio*. Так что, если вы не пишете тип типа перед названием типа, вы пишете уже не на **C**.

Но! В **C** тоже есть способо сделать так, чтобы не писать тип типа перед названием структуры. Для этого мы можем использовать `typedef`. Пример:

```
// напоминаю, что такое typedef: он позволяет создать "синоним" для типа.
// Синтаксис: typedef <старое-имя> <новое-имя>;
typedef unsigned char uchar; // теперь вместо unsigned char можно писать uchar
// объявляем наш вектор...
struct vector3 { float x, y, z; };
// Хоба! Вместо 'struct vertor3' пишем просто 'vector3'
typedef struct vector3 vector3;
vector3 Omega = {0, 0, 1};
struct vector3 omega = {1, 0, 0};
```

Но знаете, мне не нравится... Может, вспомним про анонимные типы?)

```
typedef struct { float x, y, z; } vector3;
vector3 v = {0.001, 0.001, 0};
```

Но теперь мы, к сожалению, можем писать *только без слова* `struct`. Чтобы вернуть возможность писать и так, и так, поступим следующим образом:

```
typedef struct vector3 { float x, y, z; } vector3;
vector3 v = {10, 0, -1.5};
struct vector3 a = {0, 0, -9.81};
```

Примечание: я *настоятельно не советую* пользоваться тем, что предоставляет **C++**. Да, это моё личное мнение, но структуры в **C++**, хоть и являются "надмножеством" того, чем они являются в **C**, они зачастую используются сооовсем по-другому. Если вам лень писать тип типа (и, поверьте, в этом нет ничего постыдного, даже самые используемые библиотеки в мире используют `typedef struct{...};`), просто пользуйтесь `typedef`.

P.S. Каждый раз, когда вы используете **C++** вместо `typedef struct`, в мире умирает один котёнок. По моей вине. Пожалейте бедных котят.

7 Прощальные слова

Наверное, это всё... Да, в последний код (который на всю страницу) вглядываться долго, но если написать всё на бумажке и подумать, должно быть понятно. А если непонятно, перечитайте документ. А если не помогло, то документ написан плохо. Ауот...

Save the world, my final message.
Goodbye!