Структуры

Одна из наиболее полезных возможностей C и непозволительно редко используемая начинающими программистами – возможность объединять несколько отдельных объектов в более сложные структуры данных.

В последнем задании на выделение памяти вы наверняка много времени потратили на то, чтобы корректно работать с массивами и их длинами по отдельности. Уже в игрушечном примере было легко ошибиться, о чём говорит процент успешных решений. Что же будет в больших программах?

Мы научимся использовать возможности C по структурированию данных чтобы *меньше думать* и при этом писать *более надежные* программы.

Структурный тип это пачка из данных разных типов. Опишем новый тип данных "именованной 2-D точки" `point`, в котором есть три поля: имя точки и её координаты:

```
struct point {
  const char* name;
  int64_t x;
  int64_t y;
};
```

Теперь можно создавать экземпляры типа `struct point` (не `point`! `struct` обязательная часть) и обращаться к его частям.

```
// Объявляем переменную  p1


struct point p1 = { "p1", 3, 9 };


// Изменяем поля


p1.x = 30;
p1.y = 90;
```

```
// Создаём указатель

struct point* p1_ptr = &p1;



// Одно и то же; для указателей на структуры

// есть специальный синтаксис со стрелочкой

(*p1_ptr).x = 300;

p1_ptr->x = 300;
```

В отличие от массива, где все элементы лежат в памяти последовательно без пропусков, в структурах между полями могут быть отступы. На современных компьютерах зачастую чтение из памяти происходит быстрее по адресам кратным 4 или 8, поэтому если поле структуры занимает, скажем, 3 байта, компилятор может отступить один байт до следующего поля, заполнив его мусором.

```
struct test {        // Минимум один байт пропадёт

   char first[3];    // X X X _

   int64_t second;   // X X X X

                     // X X X X

};
```

Поэтому размер структуры может быть больше, чем сумма размеров её элементов. Никогда не заменяйте корректный `sizeof(struct S)` на посчитанный вручную размер `struct S`!

Структуры могут хранить внутри себя числа, указатели, другие структуры, массивы фиксированной длины, словом — данные любых типов.

```
// Структура из двух структур, которые являются её частями

// По возможности предпочитаем такой вариант,

// а не указывать на другие структуры



struct line {

   struct point start;

   struct point end;
```

```
};


// Массив фиксированного размера внутри структуры
struct triangle {
   struct point points[3];
};
```

Структуры могут ссылаться на другие структуры через указатели. На структуры того же типа можно только ссылаться, иначе размер структуры был бы бесконечным, ведь в её состав включалась бы она сама.

```
// ломаная это точка, соединённая с ломаной меньшего размера
// ломаная также может быть пустой (NULL)
// последняя точка ломаной соединена с пустой ломаной
// const можно применять к отдельным полям структур
struct path {
   const struct point head;
   const struct path* tail;
};


// Маленькие структуры лучше передавать в функции по значению,
// то есть копировать их туда, а не передавать по указателю
void test( struct point p ) { /* ...  */ }


int main() {


  const struct point p1 = { "p1", 0, 1 };
  const struct point p2 = { "p2", 3, 5 };
```

```
  // Сделаем ломаную path1 из точек p1 -> p2
  const struct path path2 = { p2, NULL };
  const struct path path1 = { p1, &path2 };

...
}
```

***Напоминание**. Ваши собственные структурные типы, как и любые другие, нельзя снабжать суффиксом_t -- он зарезервирован для типов, объявленных в стандарте.*

Можно создать экземпляр структуры "на ходу", не объявляя переменную:

```
void test( struct point p ) { /* ...  */ }

/*
можно перечислить все поля по порядку;
если перечислены не все поля, оставшиеся инициализируются
нулевыми значениями
*/
test( (struct point) { "p1", 0, 1} );

/*
Можно (и нужно) конкретно указывать, какие поля чему равны
Остальные инициализируются нулевыми значениями.
*/
test( (struct point) { .name = "p1" } );
```

Эта запись появилась в C99 и называется *compound literal*.

Псевдонимы для структурных типов

Название типа структуры всегда включает в себя ключевое слово `struct`. Можно создать для него псевдоним с тем же или другим именем:

```
struct triangle {
    struct point points[3];
};

/*
Позволяет не писать слово struct
НО МОЖЕТ ОЧЕНЬ СИЛЬНО ЗАПУТАТЬ КОД
Обычно, так НЕ надо делать
*/
typedef struct triangle triangle;
```

Язык C всё-таки низкоуровневый, и в нём невозможно полностью абстрагироваться от управления памятью. В реальной программе нам, как правило, нужно знать, является ли тип структурным или каким-то ещё. Ключевое слово `struct` в имени типа выявляет эту информацию, хотя и выглядит тяжеловесно. Один из главных разработчиков ядра Linux по имени Greg Kroah-Hartman [считает](), что `typedef` вводит разработчиков в заблуждение относительно размеров данных (особенно если речь идёт о структурах с большим количеством вложений); как следствие, разработчики копируют большие структуры не осознавая накладных расходов на это.

```
struct point { double coord[3]; }

struct polytope {
    struct point points[1000000];
};

typedef struct polytope polytope;
//                      ^^^^^^^^^ мы создаём псевдоним с именем polytope
//
```

```
//      ^^^^^^^^^^^^^^^ для типа struct polytope


...


polytope p1;

get_polytope( &p1 );


// Одно это присваивание означает копирование

// 24 мегабайт в памяти (а если p2 локальная переменная,

// то, скорее всего, стек уже переполнится)

polytope p2 = p1;
```

Исключениями, когда `typedef` со структурными типами делать можно, будут такие ситуации:

- в структуре **всегда** будет только одно поле; см. слайд о создании новых типов для кодирования свойств данных;
- структура намеренно делается непрозрачным типом, и мы осознанно не хотим никак с ней взаимодействовать, кроме как через выделенный набор функций. Мы поговорим об этом в одном из следующих уроков.

Как функции сообщить об ошибке?

Многие функции возвращают или какое-то значение, или ошибку. Однако нельзя вернуть из функции или данные одного типа, или данные другого типа.

В прошлом мы решали эту проблему не очень элегантно:

- мы возвращали указатель на нужное число в массиве; это позволяло в случае ошибки вернуть `NULL`.
- мы использовали аргумент-указатель, чтобы фактически вернуть два значения из функции:
  - через указатель мы возвращали число;
  - в возвращаемом значении мы передавали признак того, успешно ли завершилась функция.

```
// Деление не всегда возможно, поэтому вернуть просто int64_t нельзя

// возвращаем результат через указатель result

// сама функция возвращает true если y != 0
```

```
bool divide(int64_t x, int64_t y, int64_t* result) {
   if (y == 0) { return false; }
   *result = x / y;
   return true;
}
```

Однако использование указателей сопряжено с опасностями: например, нужно следить, чтобы данные, на которые ссылается указатель, не были деаллоцированы. Такое происходит:

- когда мы возвращаем из функции указатель на её локальную переменную (данные уничтожаются после выхода из функции, а указатель остаётся).
- когда мы указываем на динамическую память, выделенную `malloc`, и освобождаем её с помощью `free`.

Более того, программа, в которой в `result` передали неправильный указатель, является синтаксически корректной, т.е. компилятор в ответ на неё не выдаст никакого сообщения об ошибке. А мы хотим, чтобы при корректном построении программы количество возможных ошибок минимизировалось.

Опциональный тип

Теперь мы покажем, как вернуть из функции "результат или ошибку" более чётко и корректно: с помощью упаковки для результата, например, для чисел типа `int64_t`. Мы упакуем результат в структурный тип, хранящий также булево значение "успех или провал". В случае успеха второе поле будет равно корректному результату, а в случае провала оно может быть равно чему угодно – нам это уже не важно.

```
// Один из двух случаев:
// - valid = true и value содержит осмысленный результат
// - valid = false и value может быть любым

struct maybe_int64 {
  bool valid;
  int64_t value;
};
```

```
// Первый случай; создаем функцию в помощь

// Не бойтесь за производительность

struct maybe_int64 some_int64( int64_t i ) {

  return (struct maybe_int64) { .value = i, .valid = true };

}



// Второй случай; можно создать не функцию,

// а константный экземпляр структуры

// Все поля инициализированы нулями

// .value = 0, .valid = false

const struct maybe_int64 none_int64 = { 0 };
```

Напишите функции вывода на экран упакованного числа и подсчёта минимума из двух упакованных чисел.

1task – program

Полные имена и эмуляция пространств имён

Заметим, что в C очень мало способов сделать функции невидимыми для других частей программы чтобы избежать конфликтов имён.

Скажем, в языке Java можно сделать несколько модулей, в каждом из которых будет своя версия функции с именем `sum`. Это могут быть, для примера, `package1.module1.doubles.sum` и `package2.module3.integers.sum`. У этих двух функций разные "полные" имена, включающие их место в структуре исходного кода, и одинаковые "краткие" имена. Однако в C концепций пакетов и пространств имён нет, поэтому в больших проектах функциям приходится давать достаточно длинные имена.

Если функция предназначена для работы со структурой, то имя этой структуры вписывается в имя функции. Так мы делали на прошлом слайде:

```
void maybe_int64_print( struct maybe_int64 i );



// Функции с массивной сигнатурой зачастую

// в коде разбивают на несколько строчек
```

```
struct maybe_int64

maybe_int64_min(

    struct maybe_int64 a,

    struct maybe_int64 b

);
```

Мы придерживаемся этого соглашения именования и далее в нашем курсе.

Другой пример использования такого соглашения:

```
mathlib_graphs_matrix_graph_int64_count();
```

Если вам кажется, что это громоздко и некрасиво, и затрудняет чтение – вы правы. Но без этого никак: это плата за то, чтобы в больших программах не умирать от сложности кода из десятков миллионов строчек и сотен тысяч имён функций и других определений.

Структура для описания массива

Задание с массивом массивов (состоящее из трёх подзаданий) формулировалось просто и не выглядело сложным для реализации. Однако кодировать его решение было неприятно, это требовало определённых когнитивных ресурсов.

Мы думаем о массивах как о сущностях с длиной, поэтому сложно работать с массивом и его длиной по отдельности. Длина это свойство массива, его характеристика, атрибут; мы не отделяем массив от его длины когда думаем о нём. Однако в программе мы вынуждены были работать отдельно с массивом и отдельно с его длиной, заводя несколько переменных. Когда же мы выделяли память под массив массивов, надо было где-то хранить ещё и их длины – в отдельном массиве, который нужно было выделять и освобождать.

Структуры снимают с нас изрядную часть когнитивной нагрузки. Когда вы работаете с набором данных как с чем-то концептуально единым, возможно, стоит объединить их в структуру.

Научимся использовать структуры чтобы работать с массивом и его длиной как с единой сущностью. Создайте структуру `array_int` из двух полей:

- `int64_t* data`, указатель на данные;
- `size_t size`, размер массива.

Добавьте функции для:

- доступа к элементу
- записи в элемент
- вывода массива
- подсчёта минимума.

2task – program

Структура для описания массива массивов

Наконец, перейдём к массиву массивов. Вместо того, чтобы хранить размеры строчек в отдельном блоке памяти, отдельно выделять под него память и освобождать её, мы теперь будем держать всю необходимую информацию вместе.

Напишите программу, которая:

- читает со входа массив массивов чисел `int64_t`. В строчках может быть разное количество элементов. Формат такой:

```
<число строк>

<количество в строке 1> <элемент11> <элемент12> ...

<количество в строке 2> <элемент21> <элемент22> ...
```

- находит минимальный элемент (один на весь массив массивов);
- вычитает M изо всех элементов массива массивов;
- выводит результат в формате:

```
<элемент11> <элемент12> ...

<элемент21> <элемент22> ...
```

Пример:

```
Исходные данные:

---

3

4 1 2 5 3

0

2 1 2

---



Результат:



---

0 1 4 2
```

```
0 1

---
```

Гарантируется, что в массиве массивов будет хотя бы одна строчка, но не гарантируется, что будет хотя бы один элемент.

Будет запущен следующий код:

```
void perform() {
  struct array_array_int array = array_array_int_read();
  struct maybe_int64 m = array_array_int_min( array );
  if (m.valid) {
    array_array_int_normalize( array, m.value );
    array_array_int_print( array );
  }
  array_array_int_free( array );
}
```

Sample Input:

```
3
9 3 2 4 54 9 2 1 872 123
8 123 12354 23 232 43412 534 8237 -99292
3 45 2 245
```

Sample Output:

```
99295 99294 99296 99346 99301 99294 99293 100164 99415
99415 111646 99315 99524 142704 99826 107529 0
99337 99294 99537
```

3task – program

Структуры как локальные переменные

Когда локальная переменная имеет тип структуры, мы можем считать, что она создаётся в стеке. Если её значение не инициализировать, оно не определено.

```
struct mystruct {
```

```
  int64_t x;
  int64_t y;
};

void mystruct_print(const struct mystruct* p) {
  printf("%" PRId64 " %" PRId64 "\n", p->x, p->y);
}

int main() {

  // Выделяем в стеке 16 байт
  // ВСЕ ПОЛЯ СТРУКТУРЫ ХРАНЯТ МУСОР
  struct mystruct var1;

  // Выделяем в стеке ещё 16 байт
  // Поля инициализированы нулями
  struct mystruct var2 = { 0 };

  mystruct_print( &var1) ;
  mystruct_print( &var2) ;
  return 0;
}
```

Возвращать значение структуры из функции тоже можно.

```
struct mystruct
mystruct_create() {
  // выделение памяти в стеке и её заполнение
```

```
  struct mystruct result = { .x = 0, .y = 10 };


  return result;  // копирование в вызывающую функцию
}


...
```

```
struct mystruct p = mystruct_create();
```

В нашей модели это будет соответствовать копированию структуры из стекового фрейма функции `mystruct_create`, где она создаётся, в вызывающую функцию. Может показаться, что это негативно скажется на производительности, но на самом деле компиляторы умеют избавляться от этого копирования.

Вместо выделения памяти под `result` компилятор может передать в функцию `mystruct_create` указатель на `p`, чтобы структура `result` создавалась прямо в том месте, куда она потом должна была бы скопироваться. Это называется *named return value optimization* и является частным случаем избавления от лишних копирований (*copy elision*). Поэтому не бойтесь возвращать из функций даже большие структуры: это избавит вас от лишних указателей и не скажется на производительности.

Структуры помогают писать более простые функции

Вообще структуры нужны чтобы описывать сущности в программе, имеющие большую сложность, чем просто числа или строчки символов. На практике, однако, не всегда ясно, какие данные объединять в структуры. Тогда полезно посмотреть на функции в вашей программе и найти те, которые принимают больше трёх-четырёх параметров. Такие функции часто являются симптомом одной из проблем в дизайне вашего кода:

- Функция делает слишком много разнородной работы. Попробуйте разбить функцию на несколько функций поменьше, каждая из которых делает что-то одно; скорее всего им потребуется меньше параметров.
- Параметры не независимы; они являются частями логически целостных сущностей, о которых вы думаете в процессе проектирования программы. Скажем, вы передаете отдельно массив и его длину, или отдельно две координаты точки. Такие сущности, возможно, стоит объединить в структуры и передавать структуры в параметрах.

Если вы следуете этим советам, качество вашего кода повысится:

- Код будет легче читать. Каждая функция даёт фрагменту программы имя, отражающее его суть.

- В коде легче искать ошибки. Часто достаточно легко установить источник ошибки с точностью до функции, и это правило работает как для больших функций, так и для маленьких. Как только функция с ошибкой обнаружена, мы её анализируем чтобы понять, что именно приводит к ошибке. Этот анализ, конечно, легче сделать для маленьких функций.
- Код легче тестировать. Для тестирования важно, чтобы функции были маленькие и получали всё необходимое через аргументы; тогда тесты легко написать.

Мы уже упоминали аппаратный стек, хранящий локальные переменные функций. Но вообще стеком можно назвать любую сущность, которая хранит элементы и поддерживает операции `push` и `pop`:

- `push` добавляет элемент на вершину стека;
- `pop` вынимает элемент с вершины стека.

Стек можно реализовать поверх массива: для этого достаточно хранить указатель на вершину стека внутри массива. Разумеется, такой стек будет ограничен размером массива, внутри которого он существует.

Для нескольких дальнейших заданий нам будет нужен такой стек внутри программы для хранения чисел, поэтому давайте реализуем его – на основе уже имеющегося упакованного массива `array_int`.

4task – program

Вы можете создавать переменные **анонимных структурных типов**. Такие типы не конвертируются в другие типы, даже если они устроены одинаково.

```
struct {
  int64_t value;
  bool    correct;
} a;

struct {
  int64_t value;
  bool    correct;
} b;

a = b ; // ошибка, разные типы (хотя и устроены одинаково)
```

```
// Можно только так

a.correct = b.correct;

a.value = b.value;
```

Обычно это используется для описания уникальных глобальных структур. Например, в микроконтроллерах бывают ситуации, когда по строго определённому адресу находятся данные, определяющие его функционирование: описание того, к каким его контактам подключены устройства; какие светодиоды включены; служебные структуры, описывающие виртуальную память и т.д.

Такие структуры бывают частями других, именованных структур. Это удобно, если "вложенный" тип больше нигде не используется:

```
struct outer {


  struct {


     int64_t a;

     int64_t b;


  } fields;


  int64_t c;

};


...



// для примера создадим экземпляр

struct outer s;
```

```
// К полям можно обращаться так

s.fields.a;

s.fields.b;

s.c;
```

В качестве примера такой структуры из реального мира приведём структуру из ядра Linux:

```
struct blk_mq_ctx {


// внутренняя структура типа, для которого нет имени
// поле этого анонимного типа называется ____cacheline_aligned_in_smp


        struct {
                spinlock_t                lock;
                struct list_head     rq_lists[HCTX_MAX_TYPES];
        } ____cacheline_aligned_in_smp;


...


        // далее идут другие поля


} ____cacheline_aligned_in_smp;
```