Лекция 5. Структуры. Разное. const.

Евгений Линский

Еще про указатели: NULL

```
1  int* p = (int*) malloc(1000000);
2  if ( p == NULL ) {
3    printf("Error: not enough memory");
4  }
```

NULL:

1. смысл: указатель ни на что не указывает
2. реализация: *#define NULL ((void*)0)* в stddef.h
3. В C++11 используйте `nullptr`

Указатели и функции

```
1  #define N 256
2
3  int* get_rand_array() {
4    int array[N];
5    for(int i = 0; i < N; i++) {
6      array[i] = rand();
7    }
8    return array;
9  }
10
11 int main() {
12     srand(time());
13     int* ra = get_rand_array();
14     printf("%d", ra[0]);
15 }
```

1. Что не так?

Указатели и функции

```
1  #define N 256
2
3  int* get_rand_array() {
4    int array[N];
5    for(int i = 0; i < N; i++) {
6      array[i] = rand();
7    }
8    return array;
9  }
10
11 int main() {
12     srand(time());
13     int* ra = get_rand_array();
14     printf("%d", ra[0]);
15 }
```

1. Что не так?
2. array — локальная переменная, после завершения функции get_rand_array она будет “удалена” со стека; использовать ее значение в main нельзя!

Указатели и функции: all ok

```
1  int* get_rand_array(int n) {
2    int* array = malloc(n * sizeof(int) );
3    for(int i = 0; i < n; i++) {
4      array[i] = rand();
5    }
6    return array;
7  }
8
9  int main() {
10     srand(time());
11     int* ra = get_rand_array(256);
12     printf("%d", ra[0]);
13     free(ra);
14 }
```

Указатели и функции: вернуть через параметр

```
1  int get_odd_array(int* src, int n, int* dst) {
2    int count = 0;
3    for(int i = 0; i < n; i++)
4      if(src[i] % 2 == 1) count++;
5    dst = malloc( count * sizeof(int) ); count = 0;
6    for(int i = 0; i < n; i++)
7      if(src[i] % 2 == 1) dst[count++] = src[i];
8    return count;
9  }
10 int main() {
11   int* result = NULL;
12   int size = get_odd_array(array, 42, result);
13   result[0] = 42;
14   free(result);
15 }
```

1. Что не так?

Указатели и функции: вернуть через параметр

```
1  int get_odd_array(int* src, int n, int* dst) {
2    int count = 0;
3    for(int i = 0; i < n; i++)
4      if(src[i] % 2 == 1) count++;
5    dst = malloc( count * sizeof(int) ); count = 0;
6    for(int i = 0; i < n; i++)
7      if(src[i] % 2 == 1) dst[count++] = src[i];
8    return count;
9  }
10 int main() {
11   int* result = NULL;
12   int size = get_odd_array(array, 42, result);
13   result[0] = 42;
14   free(result);
15 }
```

1. Что не так?
2. Адрес выделенной памяти запишется в локальную копию dst (параметры при вызове функции копируются на стек), которая будет удалена после окончания фукнции.

Указатели и функции: all ok

```
1  int get_odd_array(int* src, int n, int** dst) {
2    ...
3    *dst = malloc( count * sizeof(int) );
4    ...
5    return count;
6  }
7
8  int main() {
9    ...
10   int* result = NULL;
11   int size = get_odd_array(array, 42, &result);
12   result[0] = 42;
13   free(result)
14 }
```

Структуры

Сущность описывается набором переменных: точка в 3D, товар в программе автоматизации на складе (название, вес, количество, ...) и т.д.

```
1  struct product_s {
2    char label[256];
3    float weight;
4    unsigned int price;
5  };
6
7  struct product_s p;
8  //Почему &?
9  scanf("%s %f %d", p.label, &(p.weight), &(p.price));
10
11 struct product_s array[100];
12 array[0].weight = 42;
13
14 struct product_s* ptr = malloc(sizeof(struct product_s));
15 ptr->weight = 42;
```

Инициализация

Инициализация — присвоение начально значения.

```
1  struct product_s {
2    char label[256];
3    unsigned char weight;
4    unsigned int price;
5  };
6
7  struct product_s p = { "Milk", 100, 5 };
```

Структуры: копирование

```
struct product_s a, b;
scanf("%s%f%d", a.label, &a.weight, &a.price);
b = a;  // Полностью копирует все поля, даже массивы.
a.price += 10;
printf("%d %d\n", a.price, b.price);  // Разные значения.
```

- Оператор = требует линейное время.
- Тонкость: если в структуре есть указатель, то будет скопировано только *значение указателя*, а не данные, на который он указывает.

```
struct array_s {
  int* p;
  int n;
};
struct array_s a, b;
a.p = malloc(sizeof(int) * 100);
a.n = 100;
b = a; a.p[0] = 42;
printf("%d %d", a.p[0], b.p[0]); // Одинаковые значения
```

Структуры: копирование. Отличия от Java, Python.

В C/C++ структуру можно разместить на стеке, в глобальной памяти, на куче (malloc).

```
1  struct product_s p1;
2  struct product_s p2;
3  // p1 и p2 независимы, скопировать все поля из p1 в p2
4  p2 = p1;
5  // p3 и p4 содержат адрес одной и той же
6  // области памяти со структурой
7  struct product_s* p3 = malloc(sizeof(struct product_s));
8  struct product_s* p4 = p3;
```

Структуры: копирование. Отличия от Java, Python.

В Java/Python объект класса можно разместить только на куче (Product p = new Product()).
Java:

```
1 Product p1 = new Product();
2 // p1 и p2 содержат адрес одной
3 // и той же области памяти с объектом
4 Product p2 = p1;
5 // чтобы сделать копию нужен метод clone
```

Структуры: передача через стэк

```
1  struct product_t create_expensive(struct product_t prod) {
2      prod.price *= 2;
3      return prod;
4  }
5
6  int main() {
7      struct product_t p, expensive_p;
8      expensive_p = create_expensive(p);
9  }
```

- Как это работает?

Структуры: передача через стэк

```
1  struct product_t create_expensive(struct product_t prod) {
2      prod.price *= 2;
3      return prod;
4  }
5
6  int main() {
7      struct product_t p, expensive_p;
8      expensive_p = create_expensive(p);
9  }
```

- Как это работает?
- Надо скопирвать на стек `sizeof(product_t)` байт для параметра функции и столько же для возвращаемого значения.

Структуры: передача через стэк

```
1  struct product_t create_expensive(struct product_t prod) {
2      prod.price *= 2;
3      return prod;
4  }
5
6  int main() {
7      struct product_t p, expensive_p;
8      expensive_p = create_expensive(p);
9  }
```

- Как это работает?
- Надо скопирвать на стек `sizeof(product_t)` байт для параметра функции и столько же для возвращаемого значения.
- Это может быть долго. Поэтому, если функции по смыслу не требуется копия как в примере выше, лучше пользоваться указателями.

Структуры

```
1  void rand_fill(struct product_s* ptr) {
2    ...
3    ptr->weight = ...
4    ...
5  }
6  int main() {
7    struct product_s p;
8    rand_fill(&p); // зачем?
9  }
```

- Указатель всегда занимает одно и то же число байт вне зависимости от типа переменной, на которую указывает.
- Указатель позволяет менять значение (для структур — значение полей) переданной переменной.

Структуры: typedef

```
1 typedef struct product_s product_t; //псевдоним
2 product_t p; // меньше букв
```

- size_t — тип переменной, которая содержит размер любой сущности в памяти (например, может быть определен как *typedef unsigned long size_t*).
- В C++ можно писать сразу `product_s` вместо `struct product_s`, необходимости в `typedef` нет.
- Стандарт POSIX резервирует себе все имена, заканчивающиеся на `_t`, так что возможны пересечения и (не)компилируемость под разными Linux/macOS.

1. Чем typedef лучше define?

Структуры: typedef

```
1 typedef struct product_s product_t; //псевдоним
2 product_t p; // меньше букв
```

- size_t — тип переменной, которая содержит размер любой сущности в памяти (например, может быть определен как *typedef unsigned long size_t*).
- В C++ можно писать сразу `product_s` вместо `struct product_s`, необходимости в `typedef` нет.
- Стандарт POSIX резервирует себе все имена, заканчивающиеся на `_t`, так что возможны пересечения и (не)компилируемость под разными Linux/macOS.

1. Чем typedef лучше define?
2. Выполняется компилятором, который выяснил типы идентификаторов перед подстановкой (typedef применится только в типам, define к любому идентификатору)

const у переменной

```
1  const float pi = 3.14159;
```

Зачем?

- Компилятор проверяет, что мы не изменим *pi* по ошибке.
- Дать больше информации программисту, читающему или использующему наш код.

```
1  void  print_hex(const int a) {
2    printf("%x",a);
3  }
4  int main() {
5    int b = 4;
6    print_hex(b);
7  }
```

Программист хотел подчеркнуть, что *print_hex* не меняет параметр. Разумно?

const у указателя

const защищает то, что *перед* ним.

```
char s1[] = "hello";
char s2[] = "bye";
char const * p1 = s1;
p1[0] = 'a'; // compilation error
p1 = s2; // ok
char * const p2 = s1;
p2[0] = 'a'; // ok
p2 = s2; // compilation error
char const * const p3 = s1;
```

Но можно и так:

```
const char * p1; // equal to char const * p1;
```

const у указателя

```
1  size_t strlen(const char * s);
2  int main() {
3    char str[] = "Hello";
4    site_t s = strlen(str);
5  }
```

1. Что хотел сказать программист?

const у указателя

```
1  size_t strlen(const char * s);
2  int main() {
3    char str[] = "Hello";
4    site_t s = strlen(str);
5  }
```

1. Что хотел сказать программист?
2. Функция *strlen* не изменяет свой аргумент. Например, программист в main может не делать копию *str* перед вызовом *strlen*.