

Семестр 1. Лекция 4. Структуры. Указатели на функцию. Разное.

Евгений Линский

29 Сентября 2017

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

NULL:

- 1 смысл: указатель ни на что не указывает
- 2 реализация: `#define NULL 0` в `stddef.h`

```
#define N 256

int* get_rand_array() {
    int array[N];
    for(int i = 0; i < N; i++) {
        array[i] = rand();
    }
    return array;
}

int main() {
    srand(time());
    int* ra = get_rand_array();
    printf("%d", ra[0]);
}
```

- 1 Что не так?

```
#define N 256

int* get_rand_array() {
    int array[N];
    for(int i = 0; i < N; i++) {
        array[i] = rand();
    }
    return array;
}

int main() {
    srand(time());
    int* ra = get_rand_array();
    printf("%d", ra[0]);
}
```

- 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!

```
int* get_rand_array(int n) {
    int* array = malloc(n * sizeof(int) );
    for(int i = 0; i < N; i++) {
        array[i] = rand();
    }
}

int main() {
    srand(time());
    int* ra = get_rand_array(256);
    printf("%d", ra[0]);
    free(ra);
}
```

Указатели и функции: вернуть через параметр

```
int get_odd_array(int *src, int n, int *dst) {
    int count = 0;
    for(int i = 0; i < n; i++)
        if(src[i] % 2 == 1) count++;
    dst = malloc( count * sizeof(int) ); count = 0;
    for(int i = 0; i < N; i++)
        if(src[i] % 2 == 1) dst[count++] = src[i];
    return count;
}
int main() {
    ...
    int* result = NULL;
    int size = get_odd_array(array, 42, result);
    result[0] = 42;
}
```

❶ Что не так?

Указатели и функции: вернуть через параметр

```
int get_odd_array(int *src, int n, int *dst) {
    int count = 0;
    for(int i = 0; i < n; i++)
        if(src[i] % 2 == 1) count++;
    dst = malloc( count * sizeof(int) ); count = 0;
    for(int i = 0; i < N; i++)
        if(src[i] % 2 == 1) dst[count++] = src[i];
    return count;
}
int main() {
    ...
    int* result = NULL;
    int size = get_odd_array(array, 42, result);
    result[0] = 42;
}
```

- 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 функции.)

```
int get_odd_array(int *src, int n, int **dst) {
    ...
    *dst = malloc( count * sizeof(int) );
    ...
    return count;
}

int main() {
    ...
    int* result = NULL;
    int size = get_odd_array(array, 42, &result);
    result[0] = 42;
    free(result)
}
```


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

```
struct product_s {
    char label[256];
    float weight;
    unsigned int price;
};

struct product_s p;
//Почему &?
scanf("%s %f %d", &(p.label), &(p.weight), &(p.price));

struct product_s array[100];
array[0].weight = 42;

struct product_s *ptr = malloc(sizeof(struct product_s));
ptr->weight = 42;
```

```
void rand_fill(struct product_s *ptr) {
    ...
    ptr->weight = ...
    ...
}
int main() {
    struct product_s p;
    rand_fill(&p); // зачем?
}
```

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

- 1 Чем typedef лучше define?

Еще: `size_t` — тип переменной, которая содержит размер любой сущности в памяти (например, может быть определен как *typedef size_t unsigned long*).

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

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

Еще: `size_t` — тип переменной, которая содержит размер любой сущности в памяти (например, может быть определен как *typedef size_t unsigned long*).

Структуры: выравнивание (alignment)

```
struct dummy_s {  
    char ch;  
    int i;  
};  
printf("%d", sizeof(struct test)); //?
```

- 1 Что будет выведено?

#pragma pack(x) — задает выравнивание на x (платформозависимо).

Структуры: выравнивание (alignment)

```
struct dummy_s {
    char ch;
    int i;
};
printf("%d", sizeof(struct test)); //?
```

- 1 Что будет выведено?
- 2 Ответ: 8. Обычно процессору “удобнее” (быстрее или есть только такие инструкции) работать с адресами, выравненными (кратными) на определенный размер. Компилятор это учитывает. В примере выравнивание 4 байта, т.е. первый `ch` “дополнен” тремя нулевыми байтами.

`#pragma pack(x)` — задает выравнивание на `x` (платформозависимо).

“Универсальная” сортировка.

Задача: написать сортировку работающую с любым типом данных (int, float, product_t). Подзадачи:

- 1 передать в функцию “любой тип данных”
- 2 переставлять “любые” элементы (swap)
- 3 сравнивать “любые” элементы (cmp)

Любой тип данных: `void*`.

```
void qsort(void *array, size_t n, ...);  
  
product_t array1[100];  
int array2[20];  
qsort(array1, 100, ...);  
qsort(array2, 20, ...)
```

- ▶ `void*` — не работает адресная арифметика (почему?)
- ▶ В C++ неявное приведение типа в `void*` не вызывает ошибки (в C все можно неявно)
- ▶ В C++ требуется явное приведение типа из `void*` (в C все можно неявно)
- ▶ `void* malloc(...)`

Переставлять любые элементы: swap.

```
void qsort(void *array, size_t n, size_t elem_size, ...) {
    char *p = array;
    //смысл: swap(&array[i], &array[j])
    swap(p + i * elem_size, p + j * elem_size, elem_size);
}
void swap(char *start, char *end, size_t elem_size) {
    int i = 0;
    while( i < elem_size ) {
        ... //сами
    }
}
```

Указатель — адрес в памяти:

- ▶ В памяти хранятся переменные и двоичный код (двоичные инструкции нашей программы)
- ▶ Можно хранить адрес переменной (указатель)
- ▶ Можно хранить адрес кода (указатель на функцию)

```
void dummy(int x) {  
    printf( "%d\n", x );  
}  
  
int main() {  
    void (*func)(int);  
    func = &dummy;  
    (*func)(2);  
    // можно и так:  
    func = dummy;  
    foo(2);  
    return 0;  
}
```

Сравнивать любые элементы: cmp.

```
void qsort(void *array, size_t size, size_t elem_size,
           int (*cmp)(void *p1, const void *p2)) {
    ...
}

int cmp_int(void *p1, void *p2) {
    int *pi1 = p1; int *pi2 = p2;
    return (*pi1 - *pi2);
}

int cmp_product_by_weight(void *p1, void *p2) {
    product_t *pp1 = p1; product_t *pp2 = p2;
    return (*pp1->weight - *pp2->weight);
}

product_t array1[100];
int array2[20];
qsort(array1, 100, sizeof(array1[0]), cmp_int);
qsort(array2, 20, sizeof(array2[0]), cmp_product_by_weight)
```

- 1 Статические (*.a, *.lib): объектные файла из библиотеки присоединяются к программе в момент линковки
 - Легко установить (не нужно отдельно загружать библиотеку)
 - При выходе новой версии библиотеки (например, исправлен баг) автору программы нужно послать пользователю пересобранную версию
- 2 Динамические (*.so, *.dll): хранятся отдельно, связывание программы и библиотеки происходит в момент выполнения (помогает отдельная компонента — загрузчик)
 - Необходимо отдельно установить все необходимые библиотеки (решение: packages and repositories)
 - При выходе новой версии библиотеки (например, исправлен баг) пользователь может самостоятельно обновить только библиотеку без пересборки программы.