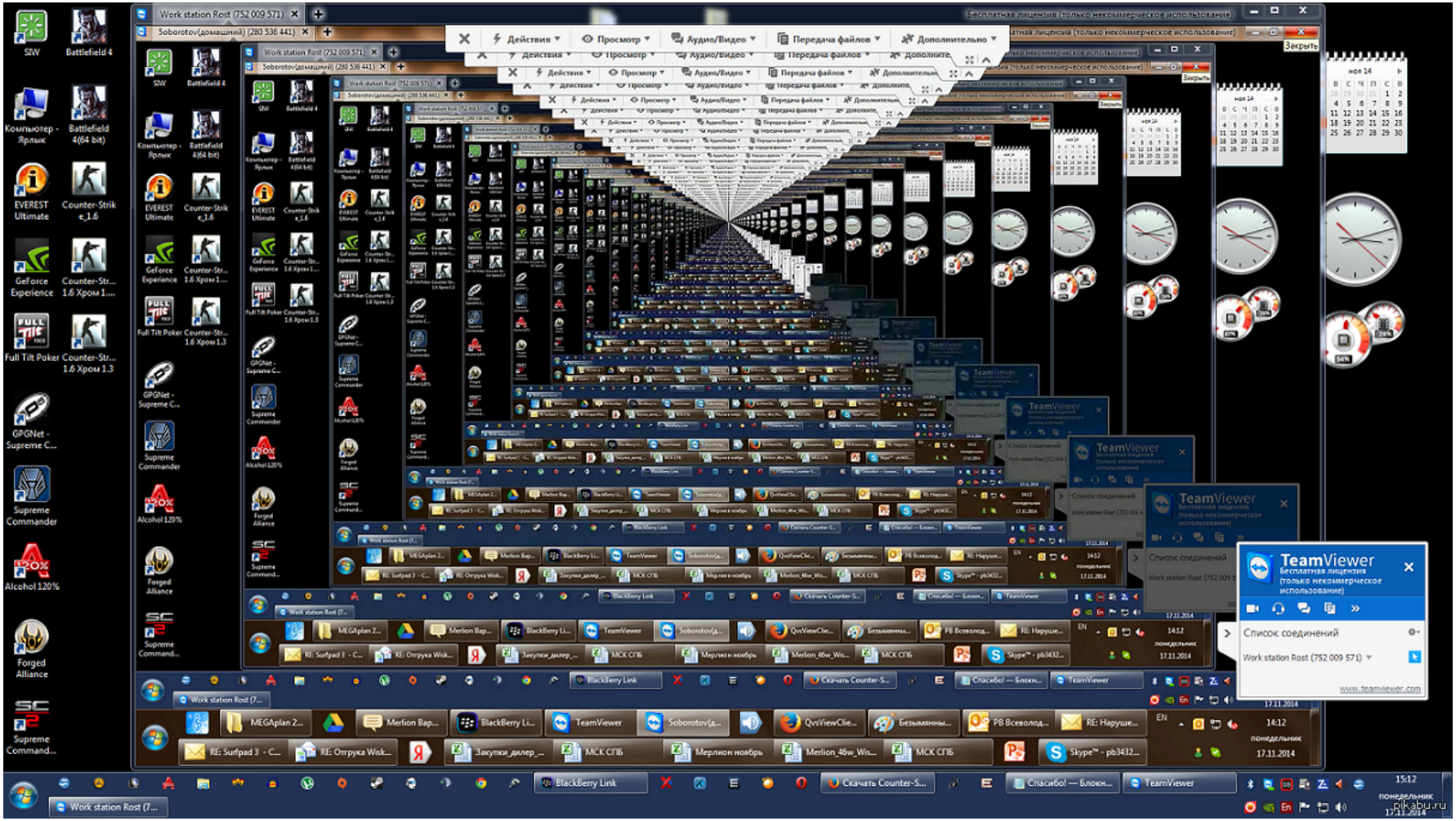
Лекция V

8 ноября 2017

Функции. Рекурсия



Функции. Рекурсия

В информатике под *рекурсией* понимается метод, в котором решение исходной задачи зависит от набора решений частных случаев этой же задачи. С для реализации подобных методов предоставляет возможность внутри функции делать вызов самой себя. Такие функции называются **рекурсивными**.

```
1 // Пример бесконечной (и хвостовой) рекурсии
2 int call_self(int num)
3 {
4   printf("Передано число: %d\n", num);
5   return call_self(num);
6 }
```

Различают *прямую* и *косвенную* рекурсивную функцию.

- **прямая рекурсивная функция** - вызывает саму себя непосредственно в своём теле
- **косвенная рекурсивная функция** - вызывает некоторую другую функцию, в теле которой происходит обратный вызов исходной функции

```
1  int start(int);
2
3  int process_number(int num)
4  {
5    return start(num);
6  }
7
8  int start(int n)
9  {
10   return process_number(n - 5);
11 }
```

Для того, чтобы быть полезной, рекурсивная функция должна включать два необходимые особенности реализации:

1. Тело функции должно обязательно содержать **условия остановки рекурсивных вызовов**.
2. Параметры функции при рекурсивном вызове **должны меняться, а не оставаться постоянными**.

Числа Фибоначчи

$$F_1 = 1, F_2 = 1, F_n(n > 2) = F_{n-1} + F_{n-2}$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, ...

Числа Фибоначчи

$$F_0 = 0, F_1 = 1, F_2 = 1, F_n(n \geq 2) = F_{n-1} + F_{n-2}$$

```
1  unsigned long long fib_num(unsigned n)
2  {
3    // условие выхода из рекурсии
4    if (n < 2) {
5      return n;
6    }
7
8    return fib_num(n - 1) + fib_num(n - 2);
9  }
10
11 printf("8-ое число Фибоначи: %llu\n",
12         fib_num(8));
```

Пользовательские типы данных

Пользовательские типы данных

Производные типы данных

Псевдонимы (aliases)

typedef

Составные типы данных

struct

enum

union (не рассматриваем)

Пользовательские типы данных. Псевдонимы

Оператор **typedef** - добавление псевдонимов для любого существующего типа данных

Завести псевдоним для типа данных:

```
typedef <тип_данных> <псевдоним1>
                [, <пс2>, <пс3>, ...];
```

для указателя на тип данных:

```
typedef <тип_данных> *<пс_ук1>,
                [*<пс_ук2>, ...]
```

С помощью псевдонимов можно создавать переменные, указывать их в параметрах функций, но в самом языке никакой новый тип данных не создаётся.

Пользовательские типы данных. Псевдонимы

Оператор **typedef**: переменные заданные через псевдоним, всё равно являются переменными исходного типа данных

```
1  typedef unsigned int uint_t;
2
3  uint_t val1 = 444;
4  unsigned int val2 = val1;
5
6  // Объявляем псевдонимы для типа double
7  typedef double velocity_t
8  // и указателя на double
9  typedef double *velocity_ptr;
10
11 velocity_t v1 = 45.5;
12 // Два варианта определения указателей
13 velocity_ptr pv1 = &v1;
14 velocity_t *pv2 = pv1;
```

Пользовательские типы данных. Псевдонимы

Оператор **typedef**: предметно-ориентированные функции

```
1  // Псевдонимы для типа и указателя на него
2  // можно записывать в одной инструкции
3  typedef double temperature_t,
4                 *temperature_ptr;
5
6  temperature_t find_start_point();
7  void make_dynamic(temperature_t start,
8                    temperature_ptr end,
9                    temperature_ptr step);
10 ...
```

Составные типы данных. Структуры

Структура - это составной тип данных, объединяющий множество *проименнованных* типизированных элементов. **Элементы структуры** называют **её полями**. Тип поля может быть любой, известный к моменту объявления структуры. Общий синтаксис:

```
struct <название_структуры>
{
  <тип_1> <поле1_1> [, <поле1_2>, ...];
  <тип_2> <поле2_1> [, <поле2_2>, ...];
  ...
  <тип_n> <полеN_1> [, <полеN_2>, ...];
} [переменная1, переменная2, ...];
```

При создании переменной структуры под каждое поле выделяется соответствующий блок памяти, и блоки располагаются в порядке объявления полей.

Использование структур: определение и объявление переменных

```
1 struct MaterialPoint
2 {
3   int x, y;
4   int z;
5   double weight;
6 };
7
8 struct MaterialPoint mp1;
```

Использование структур: объявление переменных

```
1 struct MaterialPoint
2 {
3   int x, y;
4   int z;
5   double weight;
6 };
7
8 struct MaterialPoint mp1;
```

Переменной **mp1** выделяется блок памяти, который состоит из трёх подблоков размера **int** и одного подблока размера **double**.

Использование структур: определение переменных

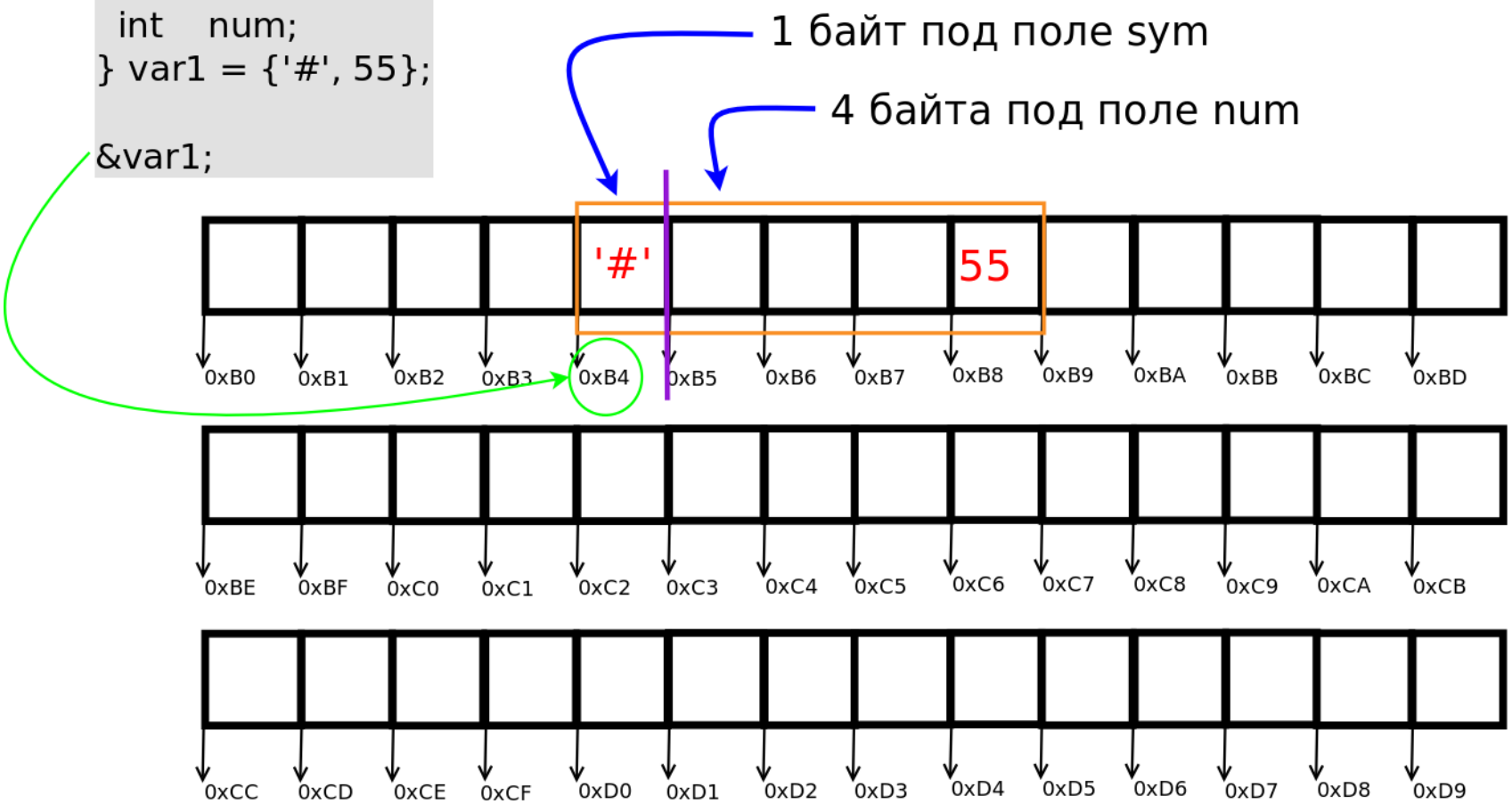
```
1 struct MaterialPoint
2 {
3   int x, y;
4   int z;
5   double weight;
6 };
7
8 struct MaterialPoint mp1 = {5, -2, 1, 5.5};
```

Для конкретной переменной стркутурного типа инициализация состоит из указания значений надлежащего типа в порядке описания полей.

Структуры: расположение переменной структуры в памяти

```
struct
{
  char sym;
  int    num;
} var1 = {'#', 55};

&var1;
```



Составные типы данных. Структуры

Использование структур: обращение к полям переменных через оператор «.»

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double weight;
5  };
6
7  // начальные значения полей не ↩
       устанавливаются
8  struct MaterialPoint mp1;
9  mp1.x = mp1.y = 4;
10 mp.weight = 45.5;
11
12 printf("Вес точки: %f\n", mp.weight);
```

Использование структур: определение структуры и объявление переменных одновременно

```
1 struct MaterialPoint
2 {
3   int x, y, z;
4   double weight;
5 } g_mp1, g_mp2;
6
7 // ...
8 g_mp1.weight = 45.3;
9 g_mp2.x = g_mp2.y = g_mp2.z = 1;
```

Использование структур: анонимные структуры

```
1  struct
2  {
3     int x, y, z;
4     double weight;
5  } g_mp1, g_mp2;
6
7  // ...
8  g_mp1.weight = 45.3;
9  g_mp1.x = g_mp1.y = g_mp1.z = 25;
10 g_mp2.x = g_mp2.y = g_mp2.z = 1;
```

Использование структур: анонимные структуры + **typedef** = избавление от **struct** при задании переменных

```
1  typedef struct
2  {
3     int x, y, z;
4     double weight;
5  } MaterialPoint;
6
7  MaterialPoint mp1, mp2;
8
9  // ...
10 mp1.weight = 45.3;
11 mp2.x = mp2.y = mp2.z = 1;
```

Составные типы данных. Структуры

Использование структур: неполное определение переменных

```
1  typedef struct
2  {
3    int x, y, z;
4    double weight;
5  } MaterialPoint;
6
7  MaterialPoint mp1 = {3, 4, 5, 8.8};
8  printf("Вес 1-ой точки: %f\n", mp1.weight);
9
10 // Можно указывать не все значения
11 MaterialPoint mp2 = {5, 8};
12 int is_zero = mp2.z == 0;
13 printf("z равно ли нулю: %d\n", is_zero);
```

При неполной инициализации поля, которым не поставлены никакие значения, получают нулевые значения.

Использование структур: определение переменных с указанием полей по именам

```
1  typedef struct
2  {
3     int x, y, z;
4     double weight;
5  } MaterialPoint;
6
7  MaterialPoint mp3 = {  .z = 3,  .y = 4,
8                         .x = 5,
9                         .weight = 8.8};
10
11 printf("Сумма x и y: %d\n", mp3.x + mp3.y);
```

Составные типы данных. Структуры

Использование структур: указатели на переменную структуры и оператор «->»

```
1  typedef struct
2  {
3     int x, y, z;
4     double weight;
5  } MaterialPoint;
6
7  MaterialPoint mp1 = { 3, 4, 5, 8.8 };
8  MaterialPoint *p_mp = &mp1;
9
10 printf("Доступ через указатель: %d\n",
11                                 (*p_mp).y);
12
13 // Получение значения поля по указателю
14 printf("И ещё раз: %f\n", p_mp->weight);
```

Использование структур: параметры функции

```
1  typedef struct
2  {
3    int x, y, z;
4    double weight;
5  } MaterialPoint;
6
7  double get_distance(MaterialPoint mp1,
8                      MaterialPoint mp2)
9  {
10   double dx = mp2.x - mp1.x;
11   ...
12   return sqrt( dx * dx + ... );
13 }
14 ...
15
16 MaterialPoint mp1 = {4, 5, 6},
17                 mp2 = {7, -4, -8};
18 printf("Расстояние: %f", get_distance(mp1, mp2));
```

Использование структур: поле-указатель на саму себя

```
1 struct DayTime
2 {
3   short hour, minute, second;
4
5   // Указателей можно добавлять сколько ↩
        угодно
6   struct DayTime *prev_moment, *next_moment;
7 };
```

Примеры структур: трёхмерный вектор

```
1  typedef struct
2  { double x, y, z; } Vector3D;
3
4  double vec_length(Vector3D v)
5  { return sqrt(v.x * v.x + v.y * v.y + v.z * v.z); }
6
7  Vector3D vec_add(Vector3D v1, Vector3D v2)
8  {
9    Vector3D v_res = {v1.x + v2.x, v1.y + v2.y,
10                     v1.z + v2.z};
11   return v_res;
12 }
13
14 Vector3D vec_sub(Vector3D v1, Vector3D v2)
15 {
16   Vector v_res = {v1.x - v2.x, v1.y - v2.y,
17                   v1.z - v2.z};
18   return v_res;
19 }
```

Примеры структур: трёхмерный вектор

```
1  double vec_sk_mult(Vector3D v1, Vector3D v2)
2  {
3    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
4  }
5
6  Vector3D vec_mult(Vector3D v1, Vector3D v2)
7  {
8    Vector3D v_res = { v1.y * v2.z - v1.z * v2.y,
9                       v1.x * v2.z - v1.z * v2.x,
10                      v1.x * v2.y - v1.y * v2.x };
11   return v_res;
12 }
```

Примеры структур: массив векторов

```
1  size_t arr_sz;
2
3  printf("Введите количество векторов: ");
4  scanf("%lu", &arr_sz);
5
6  Vector3D vec_array[arr_sz];
7
8  vec_array[0].x = vec_array[0].y = 11.5;
9  vec_array[0].z = 5.789;
10 ...
11
12 printf("Длина первого вектора: %f\n",
13         vec_length(vec_array[0]));
```

Составные типы данных. Структуры

Примеры структур: возращение более одного значения из функции (идея)

```
1  typedef struct
2  {
3    double x1, x2;
4    int is_found;
5  } SquareRoots;
6
7  SquareRoots solve_square_equation(double a,
8                                    double b, double c);
9  ...
10
11 SquareRoots roots = solve_square_equation(
12                                   15.7, -7.8, 11.3);
13
14 if (roots.is_found) {
15   printf("1-й корень: %f, 2-ой: %f",
16          roots.x1, roots.x2);
17 }
```

Перечисления (Enumerations)

Перечисления

Переисления - это пользовательский тип данных, состоящий из *конечного* набора констант, каждая из которого хранит значение **целого типа**.

Перечисления являются **открытыми (unscoped)** - каждая константа становится доступной в текущей области видимости по имени и допускается неявное приведение значений констант к числовым типам данных. Ключевое слово для объявления: **enum**

Реальный тип каждой константы может быть **char**, **int** или **unsigned int** - в зависимости от компилятора.

Перечисления

Синтаксис определения перечисления:

```
enum [<название_перечисления>]
{
  <константа_1> [= <значение>],
  [<константа_2>, <константа_3>, ...]
};
```

1. По умолчанию значение первой константы перечисления равно **нулю**.
2. Каждая константа, кроме первой, получает **на единицу большее значение**, чем предшествующая.
3. Каждой константе может быть присвоено **произвольное значение целого типа**.
4. Как только константе присваивается значение, то все следующие за ней меняются по **второму пункту**.
5. Разные константы могут иметь **одинаковые значения**.
6. Перечисления могут быть анонимными

Пример простого перечисления

```
1  enum ComputingState
2  {
3    NOT_STARTED, // значение — 0
4    STARTED,     // 1
5    COMPLETED    // 2
6  };
7
8  // Значения печатаются как числа
9  printf("NOT_STARTED: %d\n", NOT_STARTED);
10 printf("STARTED:     %d\n", STARTED);
11 printf("COMPLETED:   %d\n", COMPLETED);
```

Перечисления

Пример: использование переменных

```
1  enum ComputingState
2  {
3    NOT_STARTED = 7,    // 7
4    STARTED,            // 8
5    COMPLETED   = 11    // 11
6  };
7
8  enum ComputingState bound_task;
9  bound_task = STARTED;
10 printf("bound_task = %d\n", bound_task);
11
12 // Поля перечислений могут участвовать
13 // в числовых операциях
14 int value = (COMPLETED * 2) + STARTED;
15 int is_equals = (value == STARTED);
```

Перечисления

Пример: возращение значений из функции

```
1  typedef enum
2  {  NOT_STARTED, STARTED, COMPLETED } ComputingState;
3
4  ComputingState solve_smth(int steps, double *result)
5  {
6    ComputingState status;
7
8    if ( steps < 10 ) {
9      *result = 10.0;
10     status = NOT_STARTED;
11   } else if ( steps >= 10 && steps <= 20 ) {
12     *result = 55.873;
13     status = STARTED;
14   } else {
15     *result = 99.99;
16     status = COMPLETED;
17   }
18
19   return status;
20 }
21
22 double result;
23 ComputingState calc_state = solve_smth(25, &result);
```

Перечисления

Пример: функция для печати текстовых значений констант перечисления

```
1  typedef enum
2  { RED, GREEN, YELLOW, PURPLE } CellColor;
3
4
5  void print_color(CellColor cc)
6  {
7    switch (cc)
8    {
9      case RED      :  puts("{красный}");     break;
10     case GREEN    :  puts("{зелёный}");     break;
11     case YELLOW   :  puts("{жёлтый}");      break;
12     case PURPLE   :  puts("{фиолетовый}"); break;
13     default       :  puts("{неизвестный цвет}");
14   }
15 }
16
17 CellColor cur_color = YELLOW;
18 printf("Выбран цвет: ");
19 print_color(cur_color);
```

Немного практики: сортировка и поиск в массиве произвольного типа данных

Сортировка массивов

Сортировка массива - достаточно стандартная задача. В библиотеке <**stdlib.h**> определена функция **qsort**, с помощью которой можно сортировать массивы любого типа. Функция определена как

```
void qsort(void *elems, size_t elems_count,
           size_t elem_size, comparator);
```

- 1-ый аргумент **elems** - указатель на массив
- 2-ой аргумент **elems_count** - размер массива (количество элементов в нём)
- 3-ый аргумент **elem_size** - размер **одного** элемента массива
- 4-ый аргумент **comparator** - функция, которая умеет сравнивать **два** элемента массива

После работы функции **elems** указывает на отсортированный массив.

Сортировка массивов

Функция сравнения **comparator** должна быть определена как

```
int comparator(const void *p1, const void *p2);
```

Функция должна возвращать

- 0, если элементы на которые указывают **p1** и **p2** равны
- число меньше нуля (как правило - **-1**), если первый элемент меньше второго
- число больше нуля (как правило - **1**), если первый элемент больше второго

Сортировка массивов

В общем виде **comparator** можно представить в виде кода:

```
1  int compare_Type(const void *p1,
2                   const void *p2)
3  {
4    if ( (*(Type*)p1) <  (*(Type*)p2) )
5      return -1;
6    if ( (*(Type*)p1) == (*(Type*)p2) )
7      return 0;
8    if ( (*(Type*)p1) >  (*(Type*)p2) )
9      return 1;
10 }
```

Сортировка массивов

Пример: сортировка действительного массива

```
1  int compare_reals(const void *p1, const void *p2)
2  {
3    double val1 = *(double*)p1,
4           val2 = *(double*)p2;
5    if (fabs(val1 - val2) < 1E-8) return 0;
6    if (val1 < val2) return -1;
7    if (val1 > val2) return 1;
8  }
9
10 double my_arr[] = {55.4, 1.34, -0.95, 9.98, ↩
       43.56, 3.4};
11
12 qsort(my_arr, 6, sizeof(double), compare_reals);
13
14 for (size_t i = 0; i < 6; ++i) {
15   printf("elem %lu = %f\n", i, my_arr[i]);
16 }
```

Поиск элемента в массиве

<stdlib.h> предоставляет функцию **bsearch** поиска конкретного значения в массиве.

```
void* bsearch(const void* key, const void* elems,
              size_t elems_count, size_t elem_size,
              comparator);
```

- 1-ый аргумент **key** - указатель на переменную, содержащую значение для поиска в массиве
- 2-ой аргумент **elems** - указатель на массив
- 3-ый аргумент **elems_count** - размер массива (количество элементов в нём)
- 4-ый аргумент **elem_size** - размер **одного** элемента массива
- 5-ый аргумент **comparator** - функция, которая сравнивает значение для поиска с элементом массива

Возращаемое значение: указатель на конкретный элемент, если поиск прошёл успешно. Нулевой указатель **NULL** - иначе.

Сортировка массивов

Пример: поиск элемента в массиве

```
1  int compare_reals(const void *p1, const void *p2)
2  {
3    double val1 = *(double*)p1,
4           val2 = *(double*)p2;
5    if (fabs(val1 - val2) < 1E-8) return 0;
6    if (val1 < val2) return -1;
7    if (val1 > val2) return 1;
8  }
9
10 double my_arr[] = {55.4, 1.34, -0.95, 9.98, 43.56, 3.4};
11 double key;
12
13 printf("Введите число для поиска: ");
14 scanf("%lf", &key);
15
16 double *p_found = (double*) bsearch(&key, my_arr, 6,
17                                      sizeof(double), compare_reals);
18
19 if (p_found != NULL) {
20   printf("%f найден в массиве\n", key);
21 } else {
22   printf("массив не содержит %f\n", key);
23 }
```