# Указатели

Тема 5

# Указатель

- это переменная, которая содержит в качестве своего значения адрес памяти
- указатель может хранить адрес:
  - переменной
  - функции
  - массива
  - объекта
  - другого указателя

# Объявление указателя

Объявление указателя на целое и целого числа:

```
int *countPtr = NULL, count = 0;
```

Объявление двух указателей типа `float`:

```
float *xPtr = 0, *yPtr = 0;
```

**Хорошо!**
**NULL** - специальный макрос для обнуления указателей.
Можно также воспользоваться числом 0.

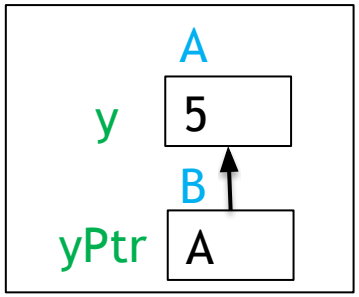
```
float *xPtr, *yPtr;
int *countPtr;
```

Плохо!
Объявлять неинициализированный указатель

# Операция адресации

- Взятия адреса или адресации **&** – унарная операция, которая возвращает адрес своего операнда

```
int y = 5;
int *yPtr = 0;
yPtr = &y;
```




- Операнд операции адресации должен быть L-величиной (т.е. чем-то таким, чему можно присвоить значение так же, как переменной)
- Операция адресации не может быть применена к константам, к выражениям, не дающим результат, на который можно сослаться

# Операция разыменования

- Разыменования или косвенной адресации * - возвращает значение объекта, на который указывает ее операнд (т.е. указатель)

```
printf("%d\n", *yPtr);                    //5
```

- Применяется только к переменным, хранящим адрес (либо к выражениям, результатом которых будет адрес)

# Операции с указателями

```
int a = 7;
int *aPtr = &a;
printf("Address a is: %x\n", &a);
printf("Value aPtr is: %x\n", aPtr);
printf("Value a is: %d\n", a);
printf("Value *aPtr is:%d\n", *aPtr);
printf("&*aPtr is %x\n", &*aPtr);
printf("*&aPtr is %x\n", *&aPtr);
```

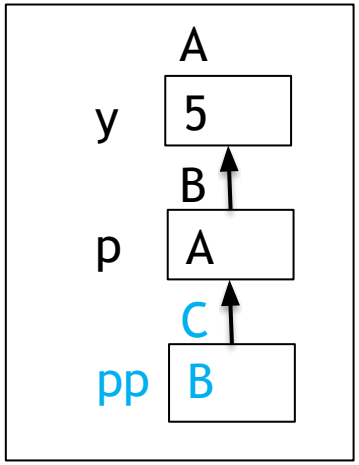
Результат:

```
0012FF7C
0012FF7C
7
7
0012FF7C
0012FF7C
```

# Указатель на указатель

- Позволяет хранить адрес переменной, хранящей адрес
- При объявлении нужно использовать две звездочки
- Для получения значения нужно использовать операцию разыменования дважды

```
int y = 5;
int *p = 0;
p = &y;
int **pp = 0;
pp = &p;
```



# Указатель на указатель

```
int a = 5;
int * p = &a;
int ** pp = &p;
printf("%d\n", a);
printf("%d\n", *p);
printf("%d\n", **pp);
printf("%x\n", &a);
printf("%x\n", p);
printf("%x\n", *pp);
printf("%x\n", &p);
printf("%x\n", pp);
printf("%x\n", &pp);
```

Результат:

```
5
5
5
0012FBA0
0012FBA0
0012FBA0
0012FB94
0012FB94
0012FB88
```

# Взаимосвязь указателей и массивов

- Имя массива – это адрес первого элемента массива
- Имя массива – это постоянный указатель
- Можно объявить указатель на первый элемент массива и использовать его вместо имени массива
- Указатель на первый элемент массива и имя массива могут использоваться практически эквивалентно

# Взаимосвязь указателей и массивов

```
#define SIZE 5
int b[5] = {1, 2, 3, 4, 5};
int* bPtr = 0;

bPtr = b;

bPtr = &b[0];
```

Эти строчки эквивалентны

```
for (int i = 0; i < SIZE; i++) {

    printf("%d\n", b[i]);

}
```

```
for (int i = 0; i < SIZE; i++) {

    printf("%d\n", bPtr[i]);

}
```

После инициализации указателя работать с массивом можно через его имя, а можно через указатель

# Арифметика указателей

Возможные действия:

```
<указатель> = <указатель> + <целое число>
<указатель> = <указатель> – <целое число>
<указатель> = <указатель> ++
<указатель> = <указатель> --
<целое число> = <указатель> - <указатель>
```

Арифметические действия с указателями имеют смысл, только если указатель ссылается на массив

# Арифметика указателей

```
#define ROW 5
double arr[ROW] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
double *p = arr;
```

| Код программы | Преобразует компилятор |
|---|---|
| `p + 3` | `p + 3 * sizeof (double)` |
| `p += 4` | `p += 4 * sizeof (double)` |
| `p - 3` | `p - 3 * sizeof (double)` |
| `p -= 2` | `p -= 2 * sizeof (double)` |
| `p++` | `p += sizeof (double)` |
| `--p` | `p -= sizeof (double)` |
| `p - arr` | `(p - arr) / sizeof (double)` |

# Арифметика указателей

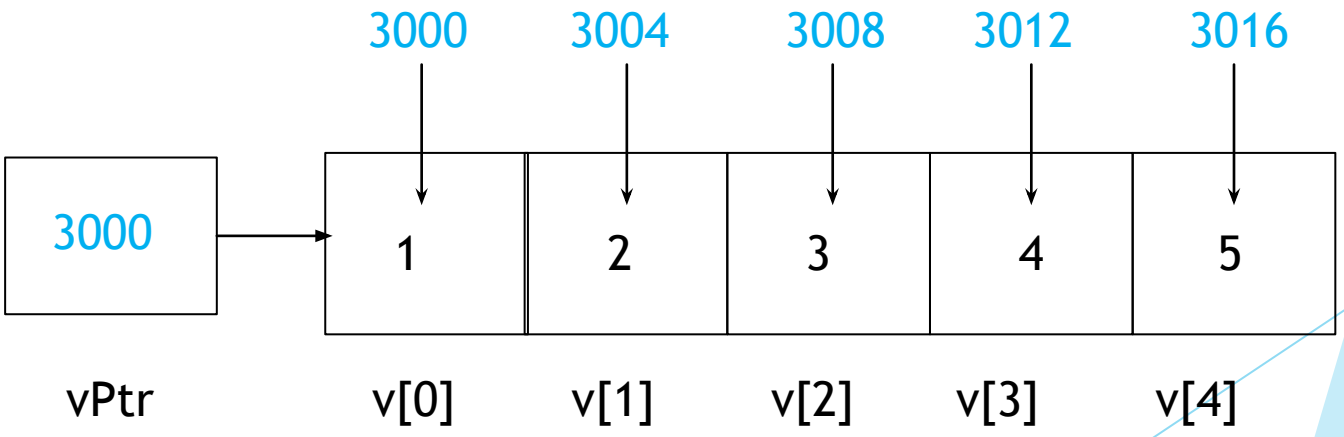
```
#define SIZE 5
int v[SIZE] = { 1, 2, 3, 4, 5 };
int *vPtr = &v[0];
printf("%x\n", vPtr++);
vPtr +=2;
printf("%x\n", vPtr);
printf("%d\n", vPtr - v);
```

Результат:

```
3000

3012

3
```



# Взаимосвязь указателей и массивов

```
int b[5] = {1, 2, 3, 4, 5};
int* bPtr = 0;

bPtr = b;

bPtr = &b[0];
```

Имя массива - это постоянный указатель, значит, оно не является L-величиной:

```
bPtr +=3;   //OK
b    += 3;  //error
bPtr ++;    //OK
b    ++;    //error
```

# Операция индексации и запись указатель-смещение

- Для доступа к элементу массива или для сдвига указателя по массиву можно использовать два варианта обращения:
  - Через операцию индексации:

```
printf("%d\n", b[3] );
bPtr[3] = 5;
```

  - Через запись указатель-смещение

```
printf("%d\n", *(bPtr+3) );
*(b+3) = 5;
```

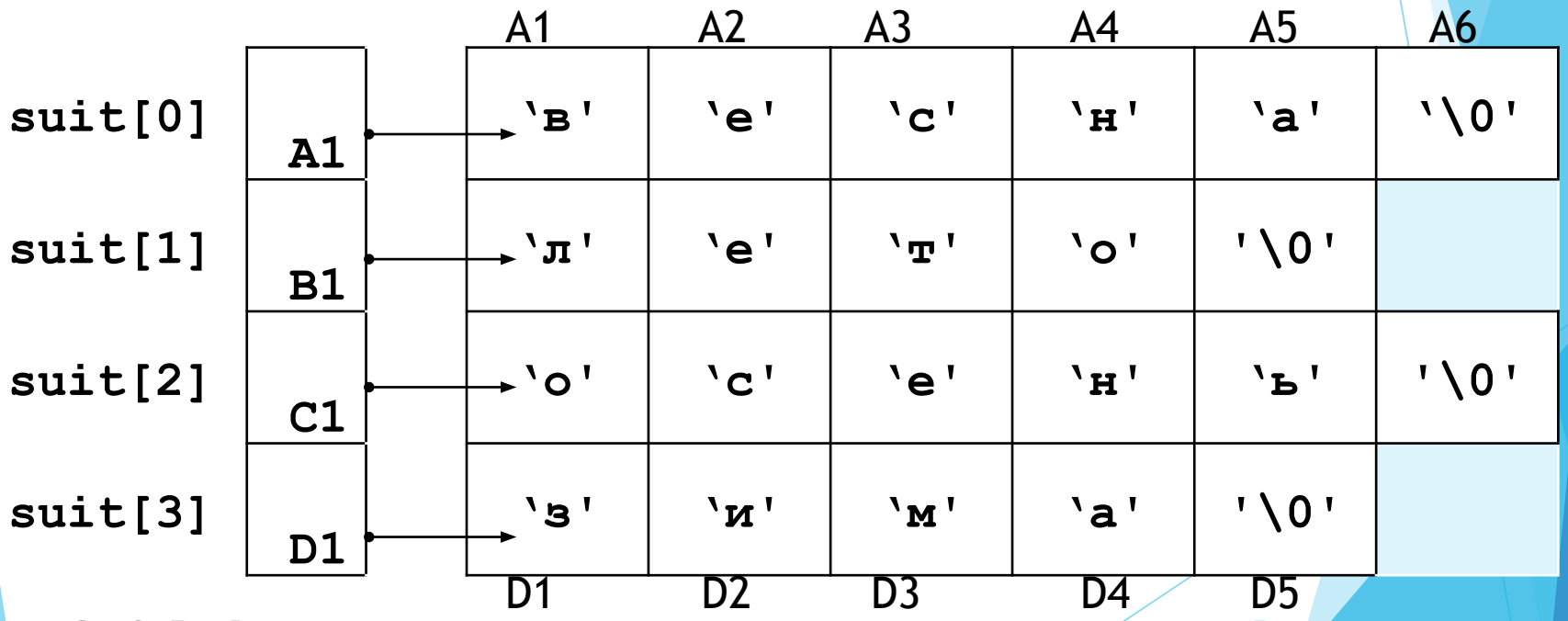
- Эти варианты эквивалентны

# Массивы указателей

- Это массивы, элементами которых являются указатели
- Используются при работе с динамическими объектами
- Указатели внутри массива могут ссылаться на массивы переменной длины
- Часто используются при работы со строками формата Си
- Массивы указателей можно рассматривать как двумерные массивы. У таких массивов известно количество строк, но не известно количество столбцов (оно может быть разным в каждой строке)

# Массивы указателей

```
char *suit[4] = { "весна",
                  "лето",
                  "осень",
                  "зима" };
```



# Массивы указателей

```
char* c[]    = {"ENTER",
                "NEP",
                "POINT",
                "FIRST"};
char **cp[]  = {c+3, c+2, c+1, c};
char *** cpp = cp;

printf("%s", * * ++ cpp);
printf("%s ", * -- * ++ cpp + 3);
printf("%s", *cpp[ -2 ] + 3);
printf("%s\n", cpp[ -1 ][ -1 ] + 1);
```

Массив указателей

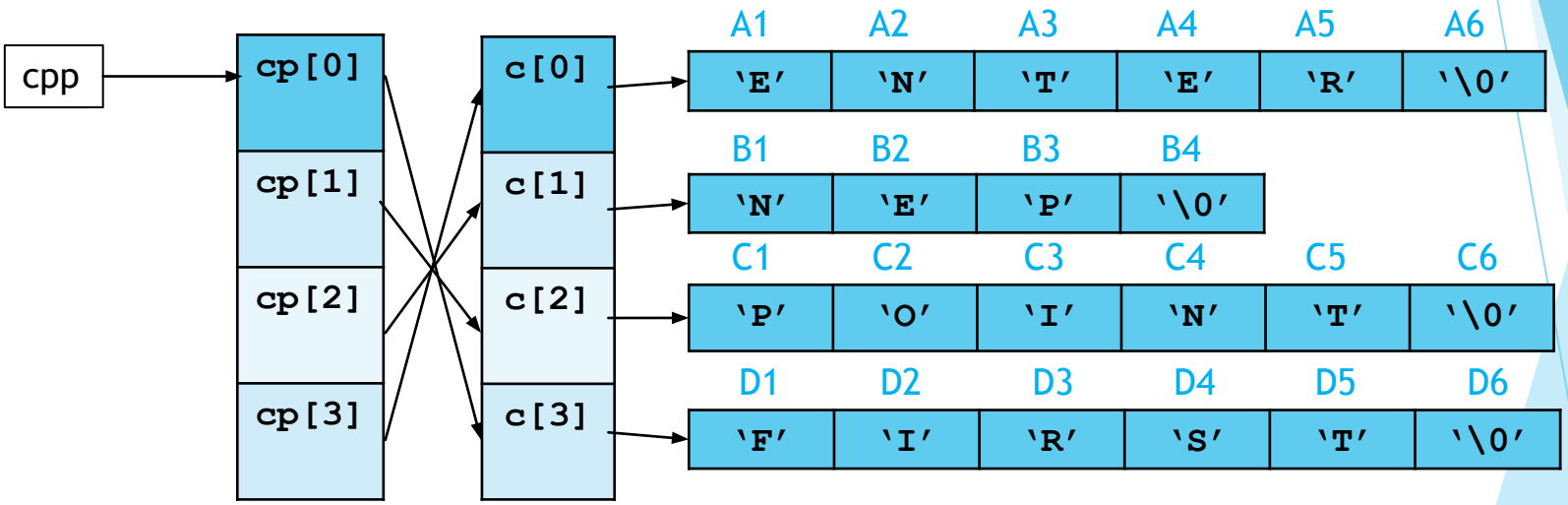
Массив указателей на указатели

Указатель на указатель на указатель

Что будет напечатано?

# Массивы указателей

Распределение памяти в задаче с предыдущего слайда



# Указатели на массивы

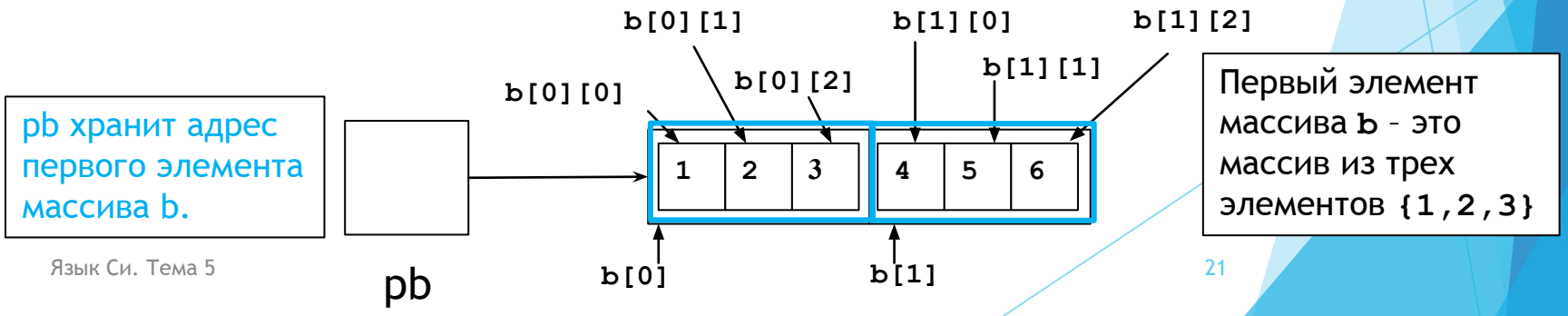
- Это указатели, которые ссылаются на целый массив, а не на отдельный элемент
- Используются при передаче многомерных массивов в функции
- При арифметике указателей смещаются на размер всего массива, на который ссылаются
- Указатели на массивы также можно рассматривать как двумерные массивы. У таких массивов может быть неизвестное число строк, но число столбцов фиксировано и не меняется

# Указатели на массивы

```
#define ROW 2
#define COLUMN 3
int b[ROW][COLUMN] = { 1,2,3,4,5,6 };
int(*pb)[COLUMN] = 0;
pb = b;
for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COLUMN; j++) {
        printf("%d\t", pb[i][j] );
    }
     printf("\n");
}
```

**pb** ссылается на первый элемент массива **b**.
Теперь через pb можно работать с массивом b

Работа как с обычным двумерным массивом




pb хранит адрес первого элемента массива b.

Первый элемент массива **b** - это массив из трех элементов **{1,2,3}**

# Указатели на массивы

```
#define ROW 2
#define COLUMN 3
int b[ROW][COLUMN] = { 1,2,3,4,5,6 };
int(*pb)[COLUMN] = 0;
pb = b;
printf("%x\n", pb);
printf("%x\n", b);
printf("%x\n", b[0]);
```

Результат:

```
0028F914
0028F914
0028F914
```

**pb** имеет тип **int(*)[3]**
**b** имеет тип **int[2][3]**
**b[0]** имеет тип **int * const**

При этом адрес, на который они ссылаются - одинаковый

# Указатели на массивы

```
#define ROW 2
#define COLUMN 3
int b[ROW][COLUMN] = { 1,2,3,4,5,6 };
int(*pb)[COLUMN] = 0;
pb = b;


printf("%x\n", (pb + 1) );
printf("%x\n", (b + 1) );
printf("%x\n", (b[0] + 1) );
```

Результат:

```
0028F920
0028F920
0028F918
```

Смещение указателей дает разные результаты:
Указатели **pb** и **b** хранят адрес массива и сдвигаются на **sizeof(int[COLUMN])**
Указатель **b[0]** хранит адрес одного целого числа и сдвигается на **sizeof (int)**

# Указатели на массивы

```
#define ROW 2
#define COLUMN 3
int b[ROW][COLUMN] = { 1,2,3,4,5,6 };
int(*pb)[COLUMN] = 0;
pb = b;
printf("%x\n", pb++);
printf("%x\n", b++);
printf("%x\n", b[0]++);
```

Результат:

```
0028F914
ошибка
ошибка
```

Указатели `b` и `b[0]` являются постоянными указателями на первый элемент массива, поэтому к ним нельзя применять операцию инкремента / декремента

# Динамические массивы

- Их размер может меняться в процессе работы программы
- Память под них выделяется и освобождается только по запросу пользователя (программиста)
- Место выделяется в специальной памяти – динамической
- Динамические массивы работают медленнее обычных (статических)

# Функции для работы с динамической памятью

- Выделение блока памяти размера **`size`**

```
void * malloc (size_t size);
```

- Выделение блока для хранения **`n`**-элементов по **`size`** байт

```
void * calloc(size_t n, size_t size);
```

- Перераспределение блока памяти

```
void * realloc(void* ptr, size_t size);
```

- Освобождение памяти

```
void free(void *ptr);
```

# Выделение памяти под динамические массивы

```
char * MyArr = 0;
int n = 0;
puts("Enter a number");
scanf("%d", &n);
//выделение памяти под массив символьного типа
MyArr =(char *) calloc(n, sizeof(char) );
…
//освобождение памяти из-под массива
free(MyArr);
```

# Выделение памяти под двумерный массив

```
int ** MyArr = 0, n, m;
puts("Enter two numbers");
scanf("%d%d", &n, &m);
//выделение памяти под двумерный массив
//сначала под массив указателей
MyArr = (int **) calloc(n, sizeof(int *) );
//потом под каждый из подмассивов
for (int i=0; i<n; i++)
{
   MyArr[i] = (int *) calloc(m, sizeof(int) );
}
```

# Освобождение памяти из-под двумерного массива

```
//сначала из-под каждого подмассива
for (int i=0; i<n; i++)
    free(MyArr[i]);
//потом из-под массива указателей
free(MyArr);
```

# Конец