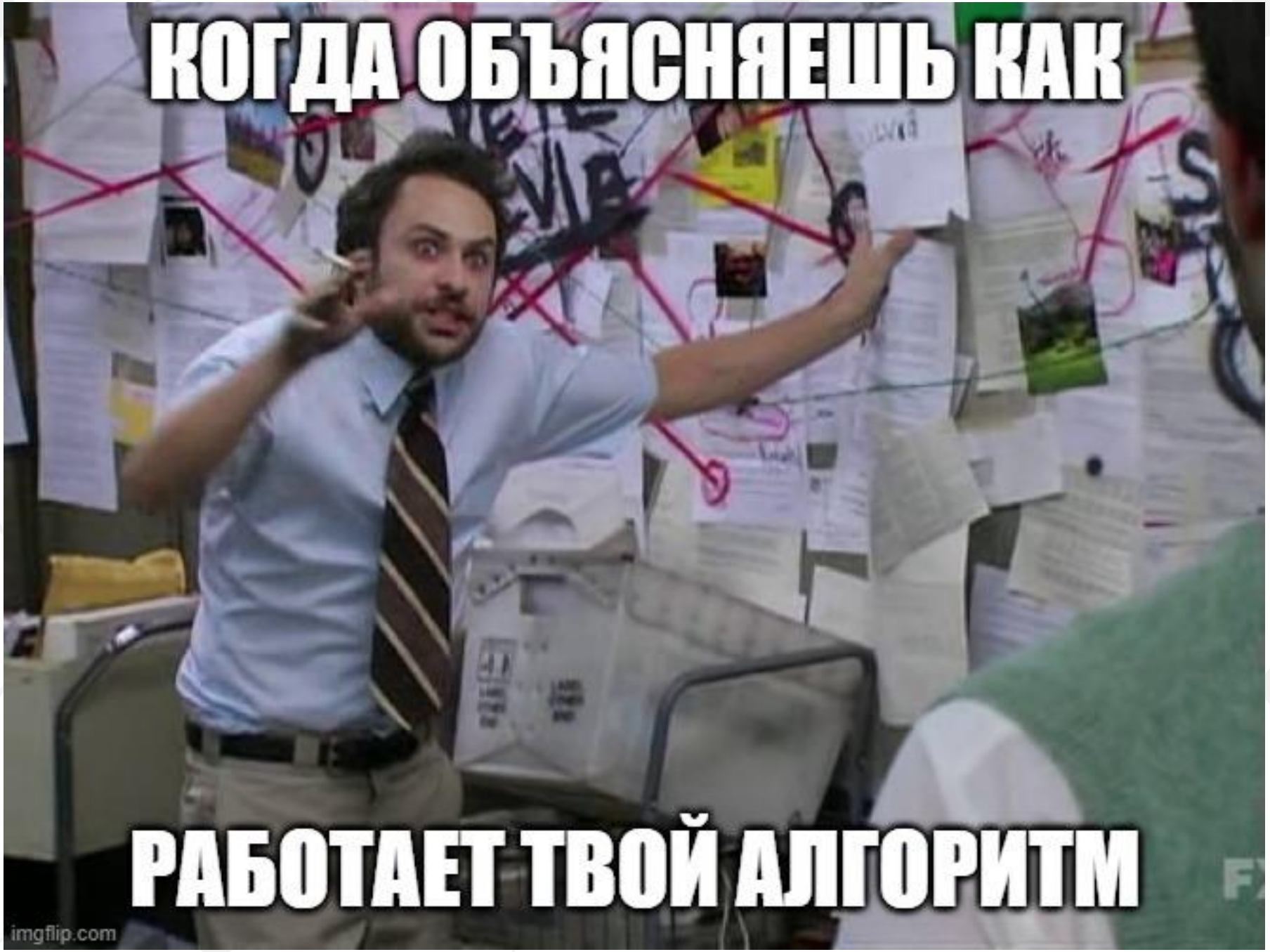
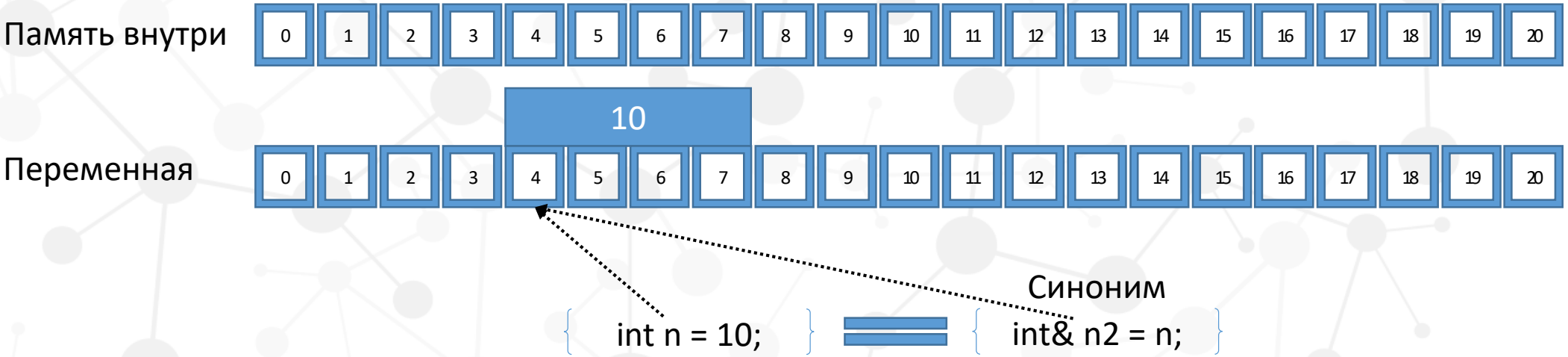
Язык программирования C++

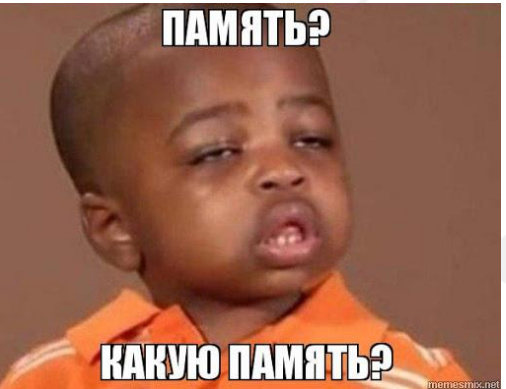
Указатели, ссылки, массивы.

Преподаватели:
Пысин Максим Дмитриевич, ассистент кафедры ИКТ
Краснов Дмитрий Олегович, аспирант кафедры ИКТ
Лобанов Алексей Владимирович, аспирант кафедры ИКТ
Крашенинников Роман Сергеевич, аспирант кафедры ИКТ


КОГДА ОБЪЯСНЯЕШЬ КАК
РАБОТАЕТ ТВОЙ АЛГОРИТМ
imgflip.com

Как работает память



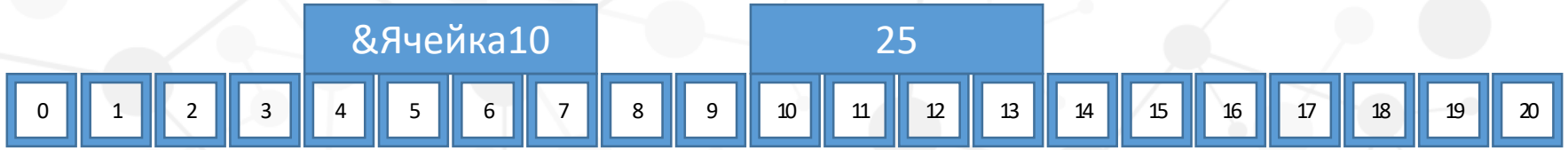


Адрес = &Ячейка

Ячейка = *Адрес

Указатель

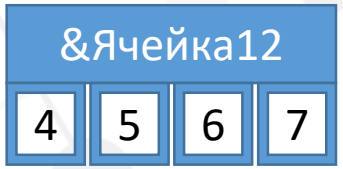
```
int n = 25;
int* pn =n&;
```




```
cout << n << endl;
cout << pn << endl;
cout << *pn << endl;
cout << &n << endl;
cout << &pn << endl;
```

```
25
&Ячейка12
25
&Ячейка12
&Ячейка6
```

Указатель



Адрес = **&**Переменная

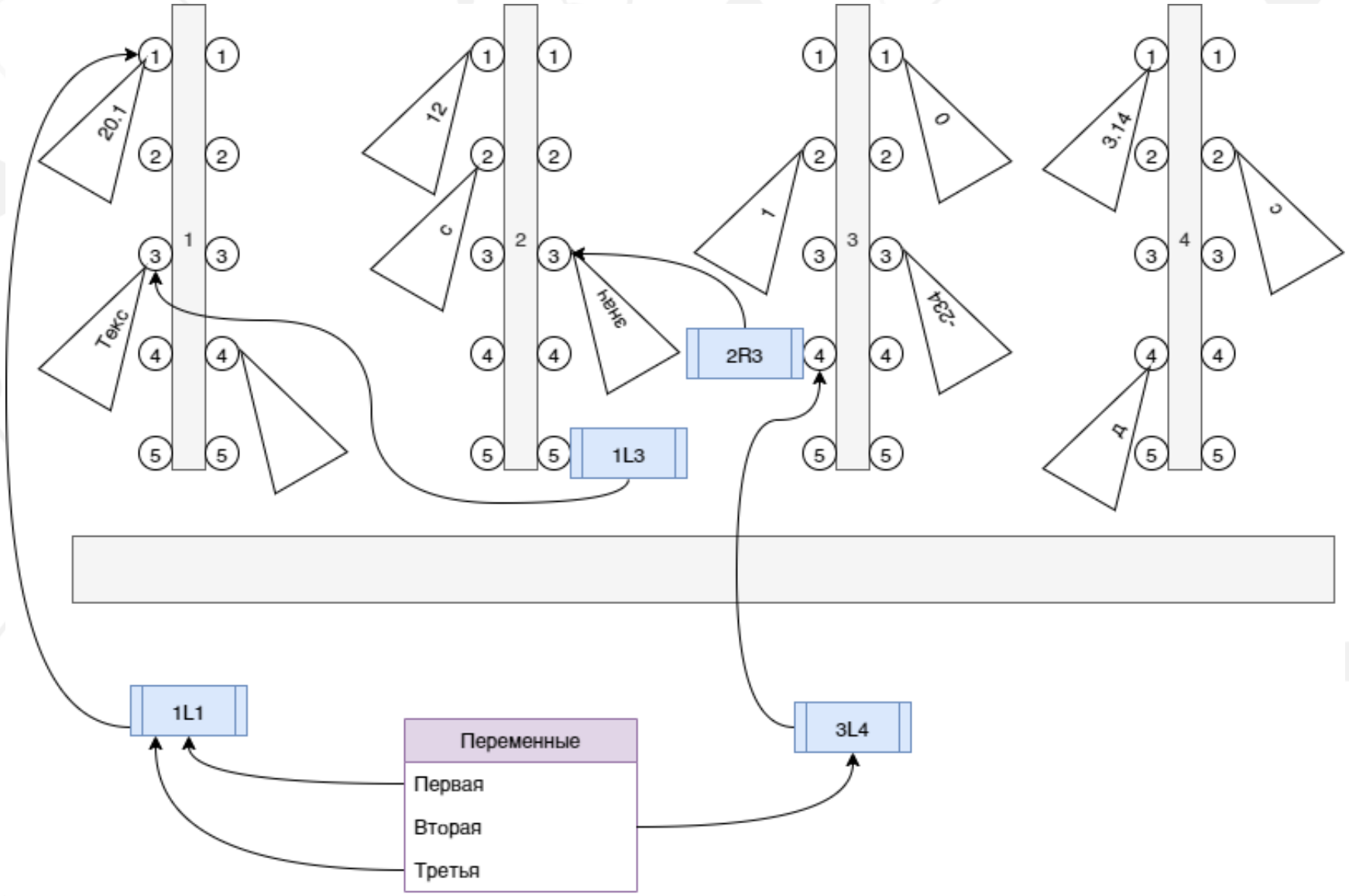
Переменная = ***Адрес**

Взятие указателя
(получение адреса)

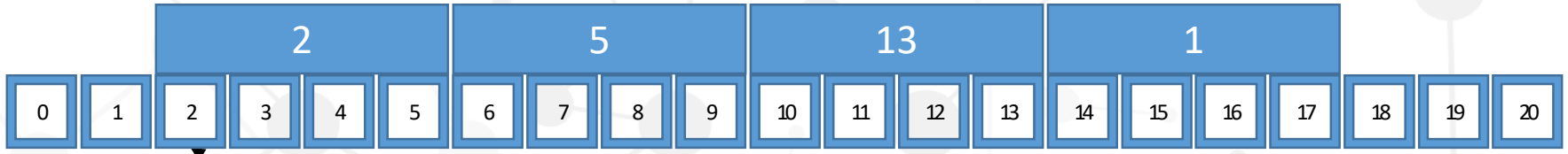
Разъименование указателя
(получение значения переменной на которую указывают)

!Указатель это отдельный тип в C++

А если проще?


20.1
Текс
12
с
2R3
1L3
1
0
1
-234
3,14
с
Д
1L1
3L4
Переменные
Первая
Вторая
Третья

Массив




```
int mass[4] {2, 5, 13, 1};
```

Но как тогда обратиться к элементу?

Переменная mass

&Ячейка2

sizeof (type/variable)
(размера переменной или типа)
Размер массива:
sizeof(array) / sizeof(array type)

Арифметика указателей

```
pointer + 1;
pointer – 2;
++pointer;
--pointer;
```

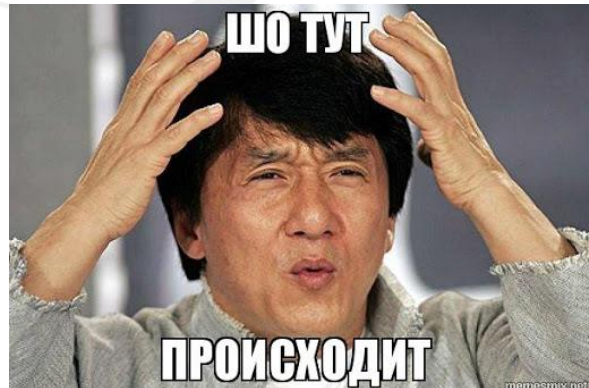
```
cout << mass[1] << endl;
```

mass[1] ⇔ *(mass + 1)

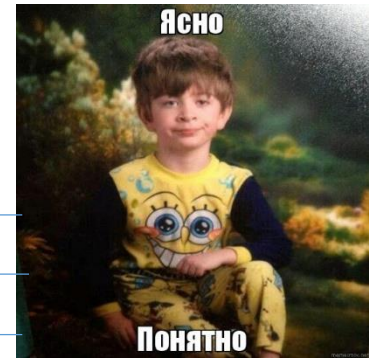
&Ячейка2 + 1 * sizeof(int) = &Ячейка6

||

&4



Пример




```
int mass[4] {3, 5, 13, 1};
int k = 256;
int* p = mass;

cout << mass[1] << endl;                          5
cout << &mass << endl;                            0x61fdf0
cout << &mass[0] << endl;                         0x61fdf0
cout << p << endl;                                0x61fdf0
cout << *(p + sizeof(int)) << endl;               6422000
cout << sizeof(int) << endl;                      4
cout << *(p + 1) << endl;                         5
cout << *mass + 10 << endl;                       13
cout << mass[0] + 10 << endl;                     13
cout << *(p + 1) + 10 << endl;                    15
*(p + 1) = 11;
cout << mass[1] << endl;                          11

void *p3 = mass;
cout << *(p + sizeof(int)) << endl;               6422000
cout << *(int*)(p3 + sizeof(int)) << endl;        5
```

Пустота, какая она?

Тип void является пустым типом, или типом пустоты, т.е. предполагается, что если компилятор встречает где либо такую запись он считает что это пустота. Используется это в объявлении функций, для указания отсутствия возвращаемого значения.
Увы, попытка создать переменную типа void обречена на провал

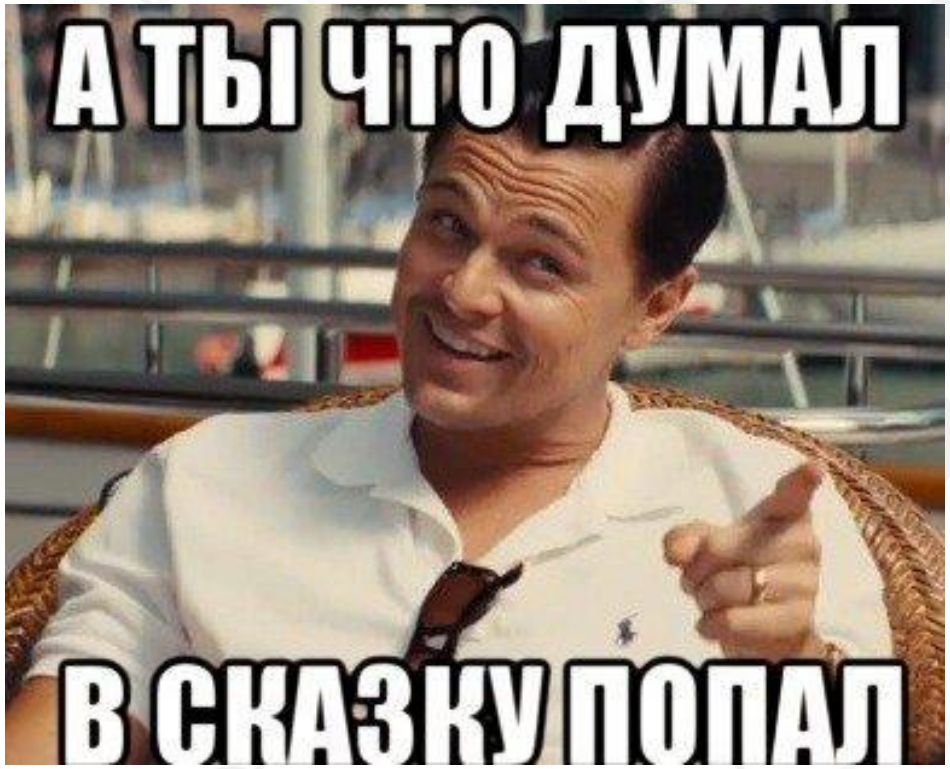
```
void function(int a){
  cout << a << endl;
}
```

Но...

```
void *p3 = mass;
cout << *(p + sizeof(int)) << endl
cout << *(int*)(p3 + sizeof(int))
```

Но существует указатель на пустоту, т.е. `void*` и вот переменную этого типа создать возможно, потому что указатель на пустоту, несмотря на свое название указывает не на пустоту, он указывает строго на 1 байт, или 1 ячейку памяти, и позволяет ссылаться на какую либо память не зная ее типа, или не обращая внимания на ее тип, а соответственно и обращаться к ней как к переменной другого типа.

Пример пустоты.




```
int buffer[10] {1,2,3,4,5,6,7,8,9,0};
int* p_buffer = buffer;
void* p_void_buffer = buffer;
cout << *p_buffer << endl;
cout << *((int*)p_void_buffer) << " -> "
     << sizeof(*((int*)p_void_buffer)) << endl;
// 1 -> 4
cout << *((long long*)p_void_buffer) << " -> "
     << sizeof(*((long long*)p_void_buffer)) << endl;
// 8589934593 -> 8
cout << *((char*)p_void_buffer) << " -> "
     << sizeof(*((char*)p_void_buffer)) << endl;
// -> 1
cout << *((bool*)p_void_buffer) << " -> "
     << sizeof(*((bool*)p_void_buffer)) << endl;
// 1 -> 1
cout << *((double*)p_void_buffer) << " -> "
     << sizeof(*((double*)p_void_buffer)) << endl;
// 4.24399e-314 -> 8
```

Разделение?

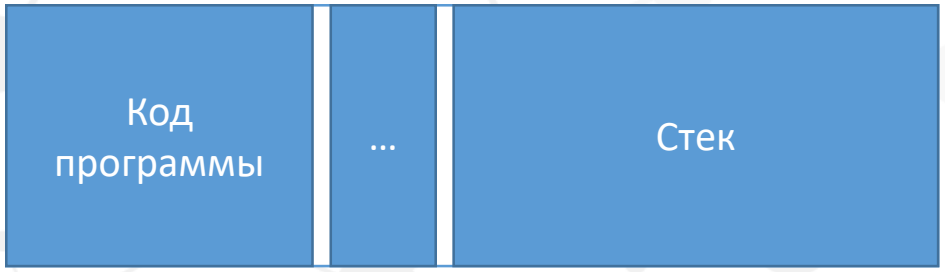
В C++ память бывает 2х видов

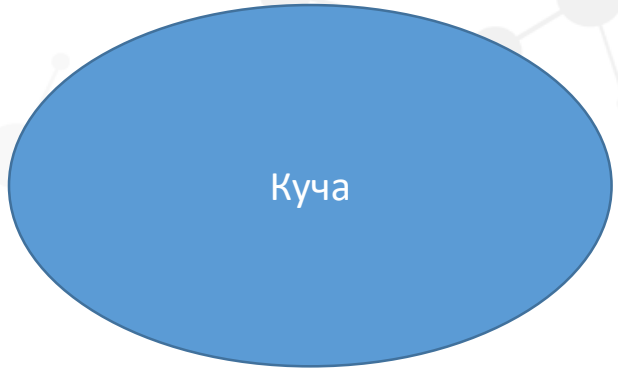
Статическая память.
Выделяется в стеке.

При статическом выделении памяти, программа самостоятельно совершает запрос на выделение, причем размер определяется автоматически на стадии запуска программы, и он достаточно мал

Динамическая память.
Выделяется в куче.

При динамическом выделении памяти программист самостоятельно создает запрос на выделение, а программа перенаправляет его операционной системе, та в свою очередь, выдает его в, так называемой, куче.





Операторы new и delete

Оператор new запрашивает выделение памяти в динамической области(куче), после чего операционная система отдает программе указатель на область памяти и устраняется от управления ей до возврата ей управления либо завершения программы.

```
<тип>* <название переменной> = new <тип>(<значение>);
```

```
int* new_int = new int(10);
float* new_float = new float;
void* new_void = new char;
```

Оператор delete освобождает память и возвращает ее управление операционной системе.

```
delete <название переменной>;
```

```
delete new_int;
delete new_float;
delete new_void;
```

Обращение к памяти по удаленному указателю чревато проблемами, но ни как не запрещается компилятором.

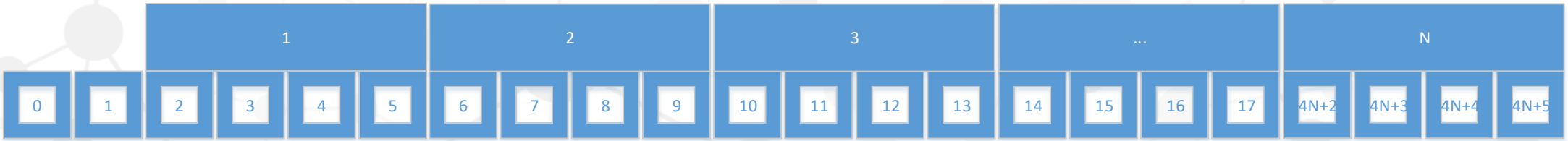
```
int * p4 = new int(1);
cout << *p4 << endl;    1
delete p4;
cout << *p4 << endl;    7817056
```

Оператор new и массив

Логично что массивы возможно выделять как в статической памяти, так и в динамической, однако в памяти они хранятся по разному:

```
int mass_stack[N];
```
Выделение памяти под массив в стеке

```
int* mass_heap = new int[N];
```
Выделение памяти под массив в куче



Визуально в памяти одномерный массив выделенный в куче и массив выделенный в стеке выглядят одинаково, это последовательно размеченная друг за другом память начинающаяся с определенной ячейки и продолжающаяся в длину столько ячеек сколько элементов содержит массив помноженный на размер типа, т.е. то сколько тип конкретного массива занимает в памяти

```
<тип>* <название переменной> = new <тип>[<количество элементов>];
```

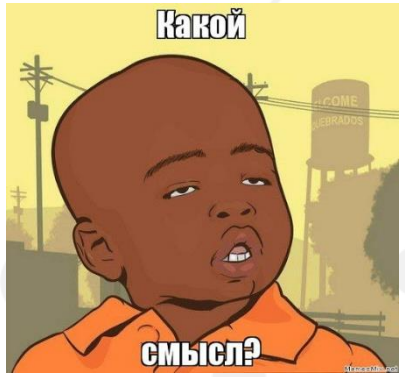
Выделение массивов больше одной размерности требует изощренности.

Оператор new и многомерный массив

Логично что массивы возможно выделять как в статической памяти, так и в динамической, однако в памяти они хранятся по разному:

```
int** mass = new int*[N];
for(int i = 0; i < N; i++)
    mass[i] = new int[N];
```

Двумерный массив в отличие от одномерного по сути является не нарезанными друг за другом блоками памяти, а отдельным набором одномерных массивов независимых друг от друга в количестве равном количеству строк в требуемом двумерном, и дополнительным массивом начала элементов каждой строки, от которых можно уже смещаться на нужную величину.



Такой извращенный способ выделения памяти позволяет создавать и использовать гораздо больше памяти в процессе работы программы, а так же делать это не на протяжении всего времени работы, а лишь в нужные моменты.

Передача параметров по указателю

При помощи указателей можно передавать параметр в функцию допуская его изменения внутри функции.

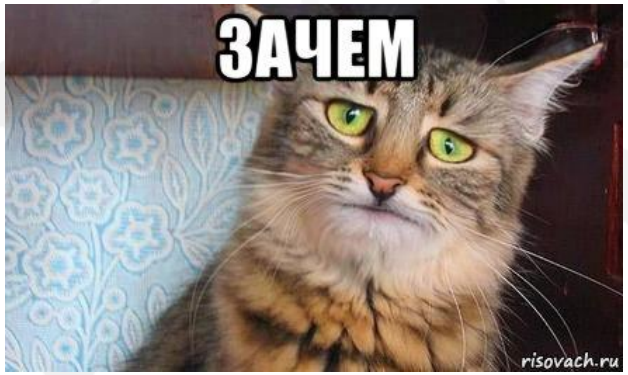
```
void add(int* to, int from){
  *to += from;
}
```

```
int from = 10;
int to = 4;
add(&to, from);
cout << to << endl; // 14
```

Так же при помощи указателей можно передавать в функцию массивы.

```
void print_array(int* array, const int N){
    for(int i = 0; i < N; i++){
      cout << array[i];
    }
    cout << endl;
}
```

```
int mass[4] {3, 5, 13, 1};
print_array(mass, 4);
// 3 5 13 1
```



Передача параметра в функцию по указателю нужна по 2 причинам.

1. Возможность изменять параметр внутри функции.
2. Исключить операции копирования параметров для оптимизации потребления памяти.

Неизменяемость и указатели

```
int n = 10;
const int N = 15;
int m = 21;
const int* p = &n;
int * const pc = &n;
int * cp = &N; // Ошибка, не дает присвоить
константу к указателю позволяющему изменять
значение
(*p) = 12; // Ошибка, не дает присвоить новое
 значение
p = &m;
(*pc) = 13;
pc = &m; // Ошибка, не дает присвоить новый у
казатель
```

Используя ключевое слово `const` мы можем сделать следующее:

`const <тип>* <название> = &<другая перменная>;`
Создать указатель который не допускает изменения значения скрытого за ним

`<тип>* const <название> = &<другая перменная>;`
Создать указатель который не допускает смены собственного значения, т.е. закрепляется за одной переменной, но позволяет ее изменять.

Наиболее часто оба варианта используется при передаче параметров в функции.

```
void test_function(const int* n, int * const m)
{
    // Блок кода
}
```

Передача по ссылке и по указателю

```
void test_function(int* n, int& m){
   // Блок кода
}
```

Передача параметров допускается как по указателю, так и по ссылке. Смысл у передачи параметра по ссылке тот же что и при передаче по указателю, исключить операции копирования. Разница в том, что при ссылки работая как синоним других переменных, не позволяют работать с динамической памятью и передавать массива.

```
int* n = new int(10);
int m {20};
test_function(n, n); // Ошибка, использовать переменную тип int* нельзя для
инициализации ссылки
test_function(&m, m); // 0x61fe14        20      0x61fe14        20
test_function(n, m); // 0xe84330        10      0x61fe14        20
test_function(&m, n); // Ошибка, использовать переменную тип int* нельзя для
 инициализации ссылки
test_function(&m, *n); // 0x61fe14        20      0xe84330        10
```

Добавим немного умного

Указатели, которые не нужно удалять –**умные указатели**. Память, выделенная под объект (переменную) будет сама освобождена, как только последний указатель на объект (переменную) выйдет за область видимости.
Для работы с ними нужно импортировать библиотеку memory

```
#include <memory>
```

unique_ptr–уникальный указатель. Нельзя присвоить второй указатель на тот же объект простым приравниванием.


```
unique_ptr<int> x_u_prt(new int(10));
unique_ptr<int> y_u_prt;
y_u_prt = x_u_prt; // Ошибка компиляции
unique_ptr<int> z_u_prt(x_u_prt); // Ошибка компиляции
int* classic_p = x_up.get(); // Превращение в обычный ук
азатель
y_up = std::move(x_up); // Передача указателя другому
y_up.reset(); // Сброс указателя
```

shared_ptr–разделяемый указатель. Можно создавать его копии (например, так удобно передать в функцию –по умному указателю). Реализует подсчет ссылок на объект. Когда последняя ссылка на объект выходит за область видимости, тогда объект будет автоматически уничтожен.

```
shared_ptr<int> x_sp(new int(42));
shared_ptr<int> y_sp(new int(13));
y_sp= x_sp; // вызовет удаление объекта с число
м 13, и оба указателя будут ссылаться на число
42
```

Проблемы delete

К сожалению операторы new и delete требуют к себе особого отношения, которое не может игнорироваться программистом на C++, ведь они является причиной множества ошибок.

Какие проблемы вызывает оператор delete:

- Можно вообще забыть вызвать delete (утечка памяти, memory leak).
- Можно забыть вызвать delete в случае исключения или досрочного возврата из функции (тоже утечка памяти).
- Можно вызвать delete дважды (двойное удаление, double delete).
- Можно вызвать не ту форму оператора: delete вместо delete[] или наоборот (неопределённое поведение, undefined behavior).
- Можно использовать объект после вызова delete (dangling pointer).

Оператор new в свою очередь может просто порождать перерасход памяти, если мы используем его в слишком больших количествах и забываем использовать delete. По этой причине, многие программисты на C++ используют определенный слой оберток над этими операторами. Один из которых это умные указатели, не позволяющие так вольно обращаться с данными.

Статические переменные

Важным упоминания при разговоре о переменных, указателях и памяти так же является ключевое слово `static` и его поведение в различных ситуациях.

```
static <тип> <название>;
```

Выделим 2 ситуации:

1. Объявляем статической переменной глобальную переменную. Тогда эта переменная перестает быть доступной вне файла в котором она определена, и будет глобальной только для него.

2. Объявляем статической переменной локальную переменную внутри функции, тогда такая переменная, в отличии от других переменных не будет уничтожаться после окончания работы функции, а будет существовать на протяжении всего времени исполнения программы. Таким образом ее значение будет сохраняться от одного вызова до другого вызова.

```
void counted_function(int k){
    static int counter = 0;
    ++counter;
   // Блок кода
}
```

Спасибо за внимание