

**Сибирь I**

**Память**

**И ВНОВЬ НЕМНОГО ООП**

# Опять работа

Нужно написать свой `string`

**В предыдущих сериях**

# Виды памяти

- *static – хранилище глобальных переменных*
- *stack – хранилище локальных переменных (и не только)*
- *heap – динамическая память (много памяти)*

**Зачем оно вообще нужно?**

**Лучше один раз увидеть**

Ordered, on top of each other



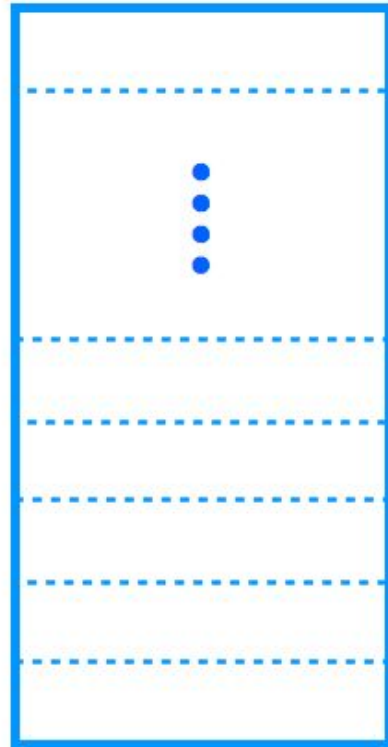
**Stack**

No particular order

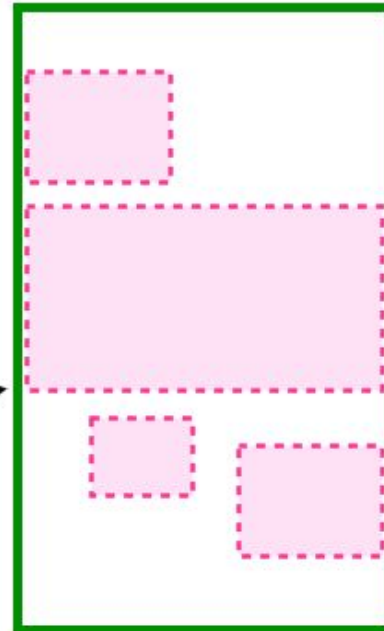


**Heap**

stack



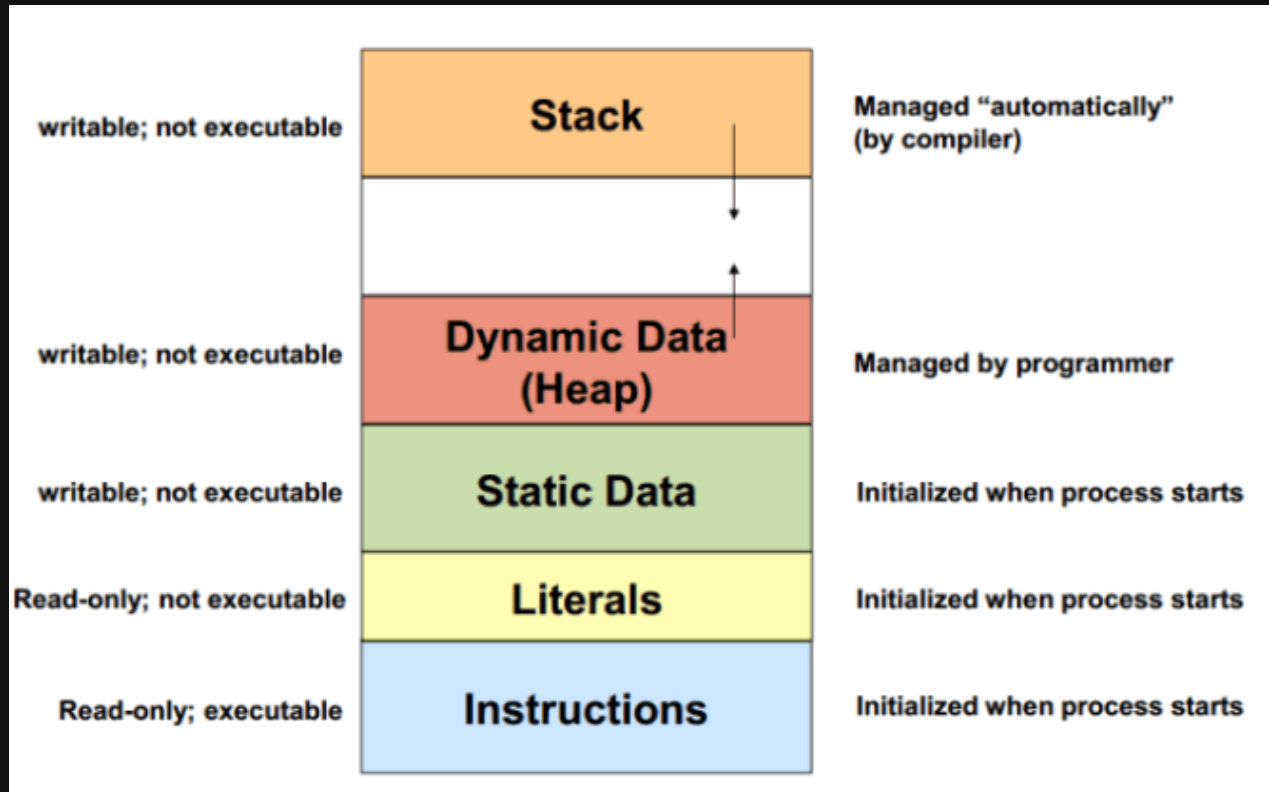
heap



static







# Микромемчик

When you mistype a keyword in C++



Static Cat



Dynamic Cat



Const Cat



Reinterpret Cat

Ever have a moment like this  
when programming?

# static memory

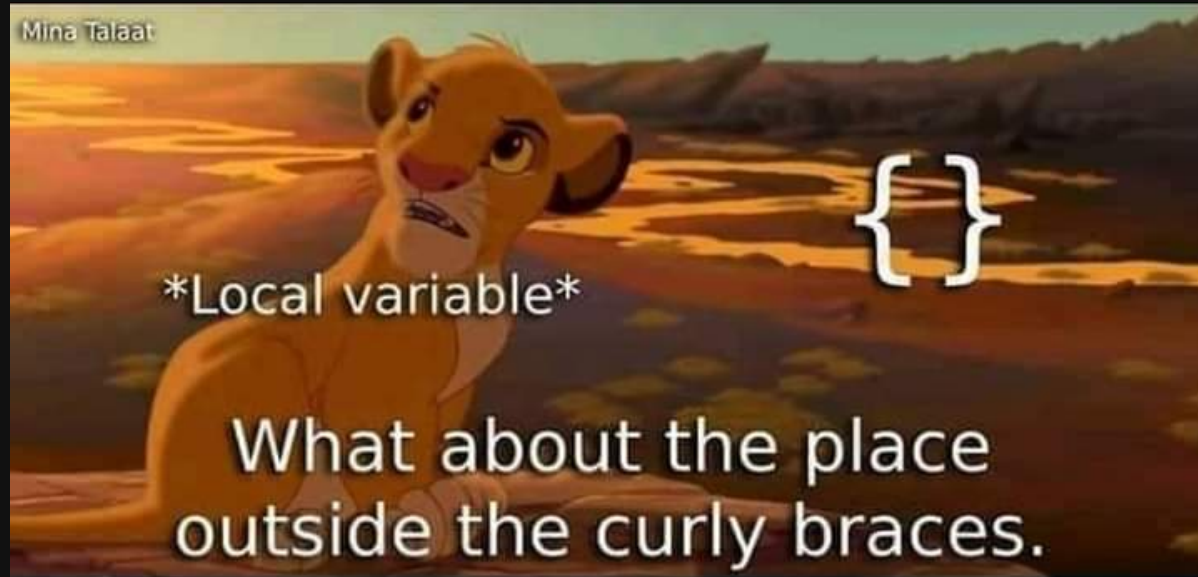
- Существует в течение всего времени работы программы
- Переменные являются статическими, существуют в единственном экземпляре и доступны в любом месте программы

# stack memory

- Контролируется CPU
- С его помощью программа вообще работает
- Переменные являются локальными
- С изменением количества переменных меняется и стек
- Стек не бесконечен

# Проверяем остаточные знания

Какова длительность “жизни” локальных  
переменных?



# heap memory

- Контролируется программистом
- `new` и `delete` – наше всё
- Для доступа требуются указатели
- Как правило ограничивается количеством доступной физической памяти

указатели



# Указатель

*Это тип данных, в котором хранится  
адрес памяти объекта (число)*

# Не путать с ссылками!

*Ссылка – альтернативное имя  
переменной*

# Примеры

```
int* p = new int(17); // создаём объект
std::cout << *p << std::endl;
delete p; // удаляем объект

int* arr = new int[n]; // выделяем область памяти
for (size_t i; i < n; ++i) { // докупаем
    p[i] = 0;
}
delete[] arr; // фиксируем прибыль
```

**Заметили ошибку?**

```
1 typedef struct s{  
2     struct s *finger;  
3 } SpiderMan;  
4 int main(void) {  
5     SpiderMan* A = new SpiderMan;  
6     SpiderMan* B = new SpiderMan;  
7     A->finger = B;  
8     B->finger = A;  
9 }
```



**Одним измерением и ограничимся?**

```
int** arr = new int*[n2]; // создаём указатель на указатель
for (size_t i; i < n2; ++i) {
    arr[i] = new int[n1](); // выделяем память
}

for (size_t i; i < n2; ++i) { // освобождаем в том же порядке
    delete[] arr[i];         // что и выделяли
}
delete[] arr;
```

**Двумя измерениями и  
ограничимся?**



**POINTER TO A POINTER TO A  
POINTER??**

**POINTER-CEPTION!**

memegenerator.net

**Трёх измерений нам  
достаточно?**

int

int \*

int \*\*

int \*\*\*

int \*\*\*\*

int \*\*\*\*\*

int \*\*\*\*\*

**И как с ЭТИМ жить?**

**Немного забегаем вперёд**

# Конструкторы и деструкторы

- **Конструктор** – блок операторов, служащий для инициализации объекта, имя совпадает с именем класса
- **Деструктор** – блок операторов, служащий для уничтожения объекта, в классах выглядит:

~ClassName

# RAII

*Resource Acquisition is Initialisation*

*Получение некоторого ресурса  
неразрывно совмещается с  
инициализацией, а освобождение – с  
уничтожением объекта*

**Пользуемся не только  
конструкторами**



# std::unique\_ptr

Облегчает работу с указателями

```
{
    {
        std::unique_ptr<int[]> vec_ptr(new int[3]{1, 2, 3});
        std::cout << vec_ptr[0] << vec_ptr[1] << std::endl;
    }
    // vec_ptr очищен
    ...
}
```

Осуществляет единственное владение!

void \*

