

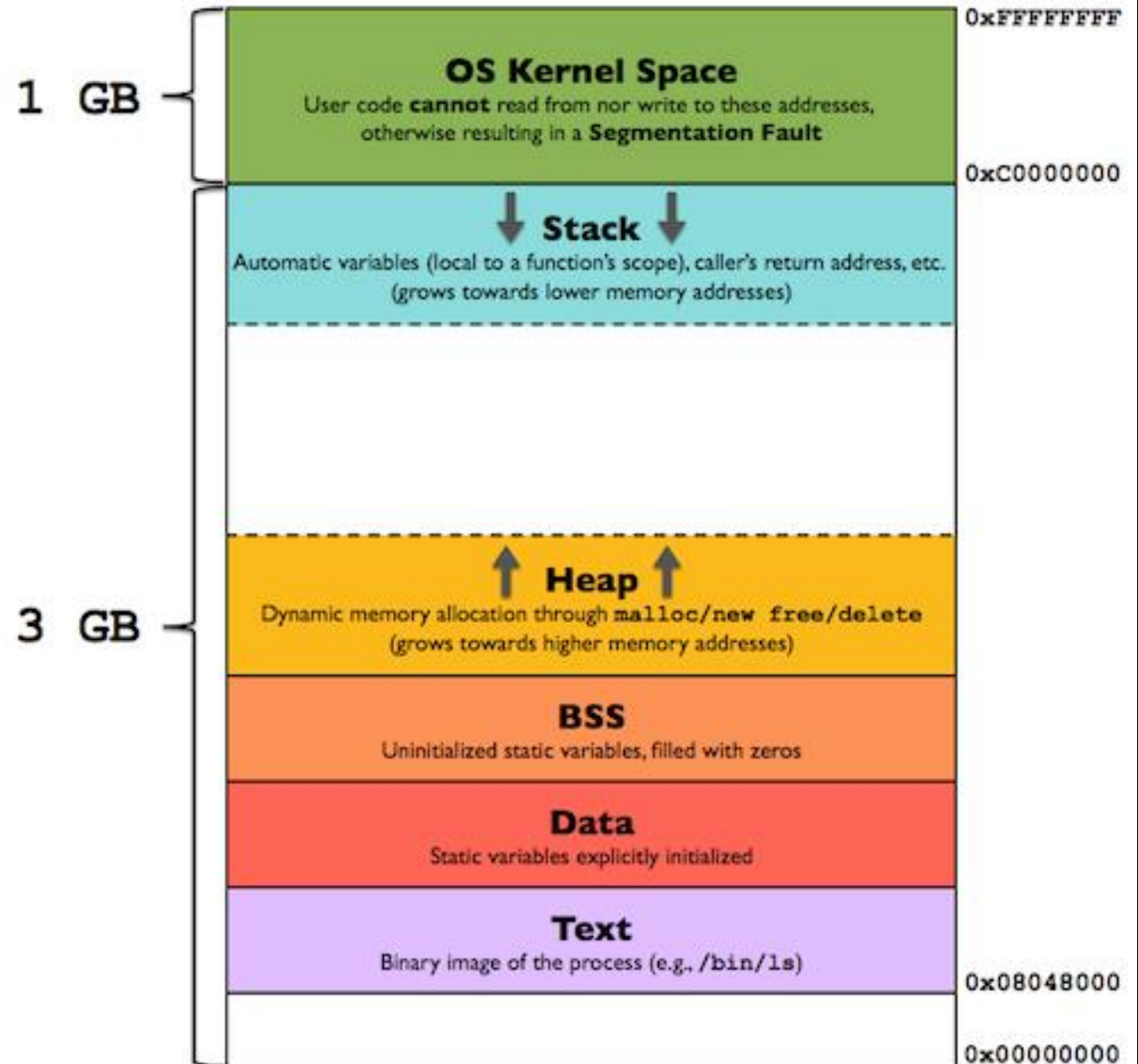
MEMORIZE

Минимум о памяти: стек, куча, глобальная память

К. Владимиров, Intel, 2018
mail-to: konstantin.vladimirov@gmail.com

Виды памяти

- Runtime stack (стек исполнения)
 - локальные переменные
 - аргументы функций
 - служебная информация (адреса возврата)
- Глобальные данные
 - глобальные переменные
 - тела функций
 - всё, что видно для objdump
- Heap (куча)
 - всё самое интересное



Три вида памяти в вашей программе

- Программа, содержащая все три вида переменных

```
int x[100];  
int main () {  
    int y[100] = {0};  
    int *pz = calloc(100, sizeof(int));  
    free(pz);  
    return 0;  
}
```

Три вида памяти в вашей программе

- Глобальная память: переменная выделяется в data/rodata

`int x[100];`  глобальный массив

```
int main () {  
    int y[100] = {0};  
    int *pz = calloc(100, sizeof(int));  
    free(pz);  
    return 0;  
}
```

Три вида памяти в вашей программе

- Куча: переменная явно выделяется вызовом `calloc` или `malloc`

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```

```
    int *pz = calloc(100, sizeof(int));
```



массив в куче

```
    free(pz);
```

```
    return 0;
```

```
}
```

Три вида памяти в вашей программе

- Куча: переменная явно освобождается вызовом free

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```

```
    int *pz = calloc(100, sizeof(int));
```

```
    free(pz);
```

```
    return 0;
```

```
}
```



явное освобождение

Три вида памяти в вашей программе

- Стек: переменная выделяется в **стековом фрейме**

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```



массив на стеке

```
    int *pz = calloc(100, sizeof(int));
```

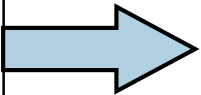
```
    free(pz);
```

```
    return 0;
```

```
}
```

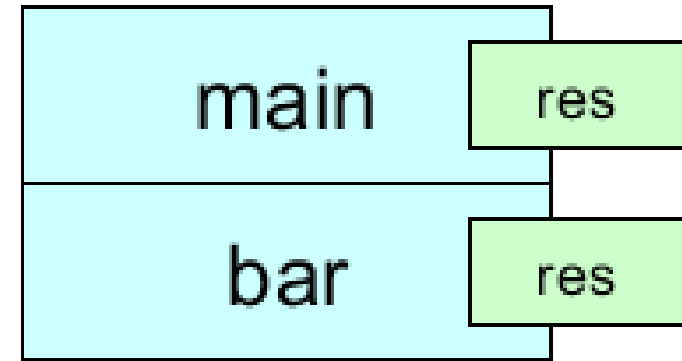
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



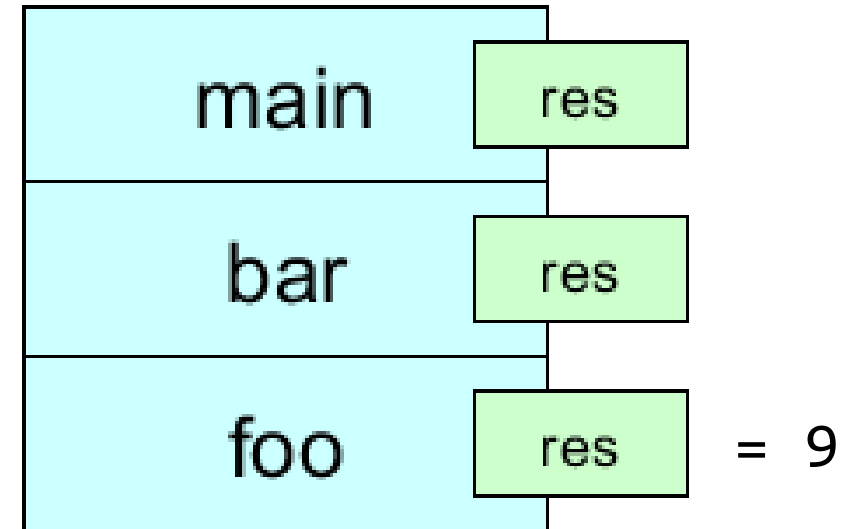
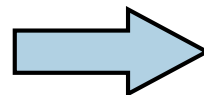
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



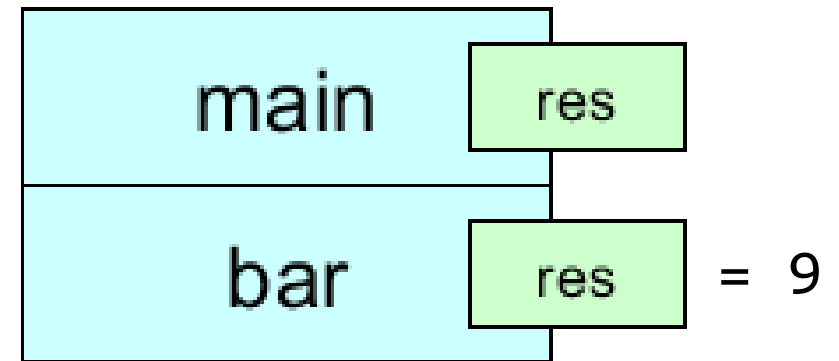
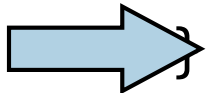
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



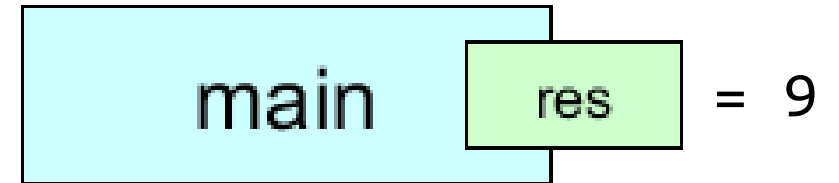
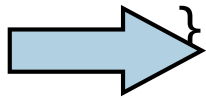
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



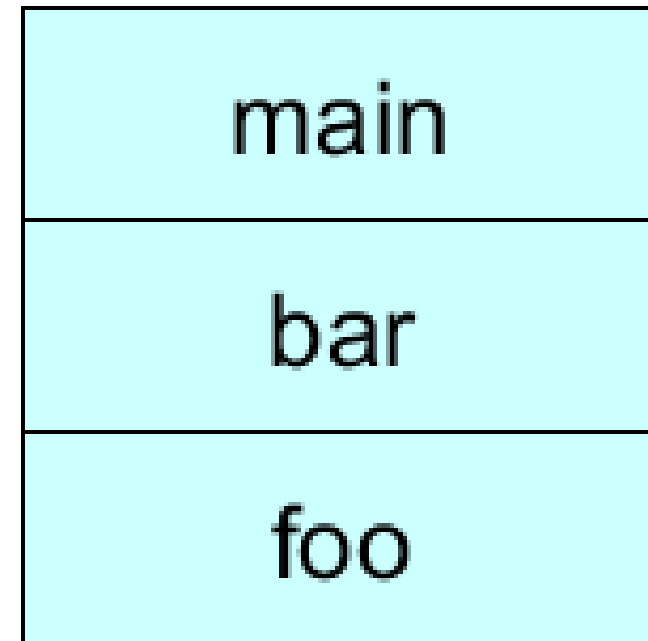
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



Почему стек называется стеком

- Вызовы функций образуют естественную LIFO структуру (собственно стек)
- Этот стек также называется стеком исполнения ([runtime stack](#))
- Все переменные, которые определены локально внутри функции, занимают её [стековый фрейм](#)
- Они также называются [автоматическими переменными](#)



Проблемы с временем жизни

- Функция возвращает адрес автоматической переменной

```
int *foo(int x, int y) {  
    int res = x + y;  
    return &res;  
}
```

- Этот адрес используется, но того места на стеке, куда он указывал, уже не существует

```
int main() {  
    int *p = foo(0, 42);  
    printf("%d\n", *p); // oops!  
}
```

- Никогда так не делайте

Работа с кучей: malloc и free

- Функция возвращает адрес в куче

```
int *foo(int x, int y) {  
    int *res = malloc(sizeof(int));  
    *res = x + y;  
    return res;  
}
```

- Этот адрес может быть свободно использован в вызывающей функции

```
int main() {  
    int *p = foo(0, 42);  
    printf("%d\n", *p); // ok  
    free(p);  
}
```

Минимум о динамической памяти

- Выделение `malloc` (размер) и `calloc` (количество, размер элемента)
- Специальный тип `void*` означает "указатель на нетипизированную память" приводится к любому указателю

```
void *mem = malloc(10 * sizeof(double)); // 10 doubles
double *pd = (double *) mem;
int *pi = (int *) calloc(1000, sizeof(int));
```

- С возвращённым указателем, можно работать как с массивом

```
assert (*pi == pi[0]); pi[100] = 2; pd[3] = 4.0;
```

- Освобождение `free` (указатель) при этом `free(NULL)` это ок

```
free(mem); // или free(pd) но не вместе
```


Проверка результата

- Исчерпание кучи гораздо менее болезненно, чем исчерпание стека. В этом случае malloc просто вернёт NULL и это можно явно проверить.

```
int *foo(int x, int y) {  
    int *res = malloc(sizeof(int));  
    assert (res != NULL); // или более сложная обработка  
    *res = x + y;  
    return res;  
}
```

- Допустимо не приводить явно, например строчка выше эквивалентна

```
int *res = (int *) malloc(sizeof(int));
```

Problem M1: исследование кучи

- Напишите программу которая будет выделять всё большие по размеру блоки памяти с помощью malloc
- Начните с десяти байт и удваивайте каждую итерацию
- На какой итерации malloc вернёт NULL?

Problem M2: исследование стека

- Напишите программу, которая будет рекурсивно вызывать функцию, создающую большой массив на стеке (например 10000 байт)
- За каждый уровень рекурсии печатайте на экран следующее число
- Сколько стека вы сможете использовать до возникновения проблем?

Область видимости и время жизни

- Глобальная переменная: область видимости везде время жизни всегда
- Переменная на стеке: область видимости в пределах блока, время жизни в пределах блока

```
int foo () {  
    int x;  
    int *py;  
    {  
        int y = 5;  
        py = &y;  
    } ← конец времени жизни y  
    x = *py; // BOOM  
} ← конец времени жизни x
```

Область видимости и время жизни

- Глобальная переменная: область видимости везде время жизни всегда
- Переменная в куче: область видимости в пределах блока, время жизни до явного освобождения

```
int foo () {  
    int x;  
    int *py;  
    {  
        int* pz = malloc(sizeof(int)); *pz = 5;  
        py = pz;  
    }  
    x = *py; // ОК  
}
```

 утечка памяти под py

Статические переменные

- Статическая переменная это глобальная переменная с ограниченной областью видимости

```
int foo() {  
    static int x = 0; // время жизни: всегда  
    x += 1;  
    printf("%d\n", x);  
}
```

```
foo(); // на экране 1
```

```
foo(); // на экране 2
```

```
foo(); // на экране 3
```