Схема распределения памяти в программе на Си: стек

Особенности использования локальных переменных

Для хранения локальных переменных используется так называемая автоматическая память.

“+”

- Память под локальные переменные выделяет и освобождает компилятор.

“-”

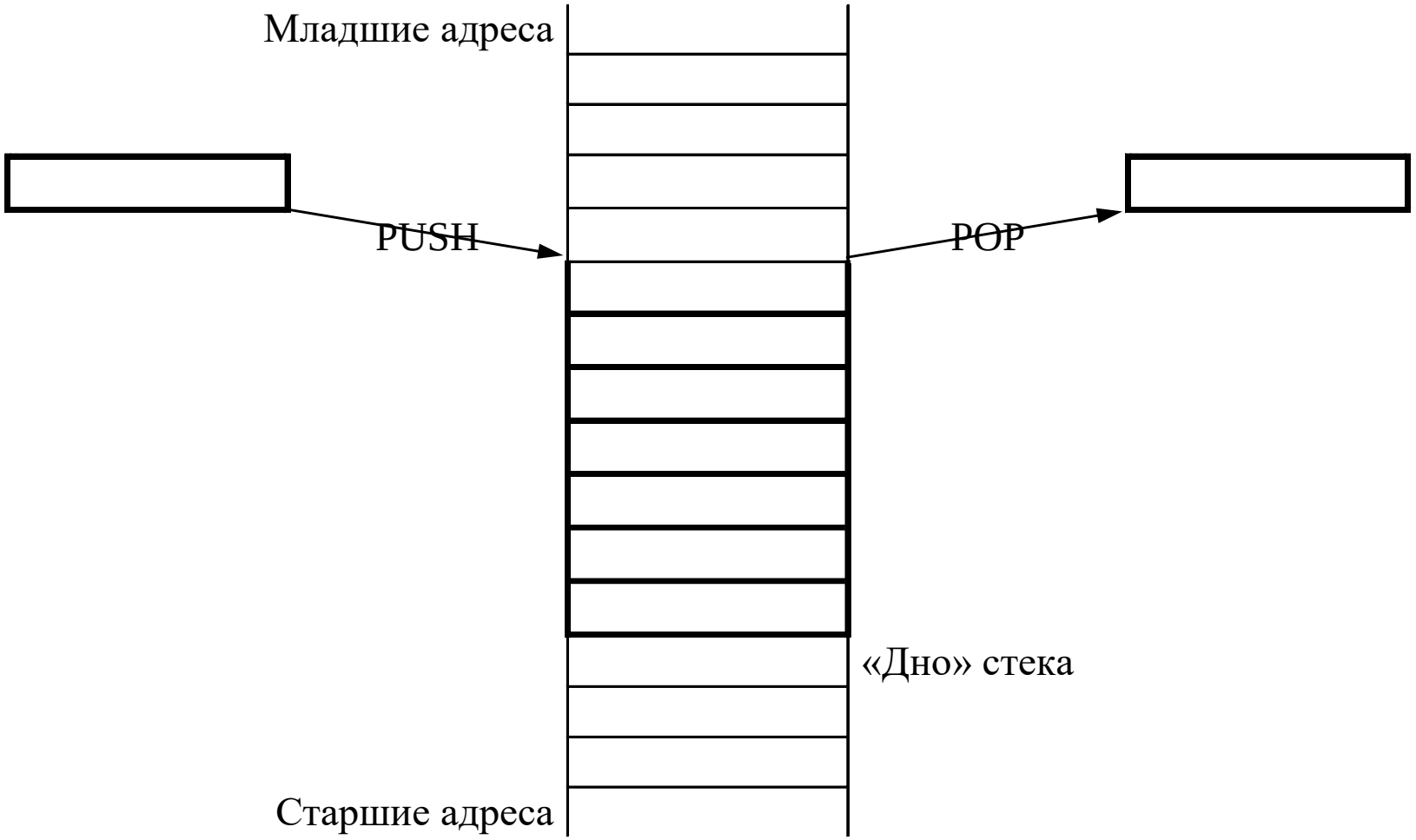
- Время жизни локальной переменной "ограничено" блоком, в котором она определена.
- Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции.
- Размер автоматической памяти в большинстве случаев ограничен.

Организация автоматической памяти

```
void f_1(int a)
{
    char b;
    // ...
}
void f_2(double c)
{
    int d = 1;
    f_1(d);
    // ...
}
int main(void)
{
    double e = 1.0;
    f_2(e);
    // ...
}
```

1. **Вызов main**
2. Создание e
3. **Вызов f_2**
4. Создание c
5. Создание d
6. **Вызов f_1**
7. Создание a
8. Создание b
9. **Завершение f_1**
10. Разрушение b
11. Разрушение a
12. **Завершение f_2**
13. Разрушение d
14. Разрушение c
15. **Завершение main**
16. Разрушение e

Стек



Использование аппаратного стека

1. вызова функции

   call name

   - поместить в стек адрес команды, следующей за командой call
   - передать управление по адресу метки name

2. возврата из функции

   ret

   - извлечь из стека адрес возврата address
   - передать управление на адрес address

Использование аппаратного стека

3. передачи параметров в функцию

соглашение о вызове:

- расположение входных данных;
- порядок передачи параметров;
- какая из сторон очищает стек;
- etc

cdecl

- аргументы передаются через стек, справа налево;
- очистку стека производит вызывающая сторона;
- результат функции возвращается через регистр EAX, но …

Использование аппаратного стека

4. выделения и освобождения памяти под локальные переменные

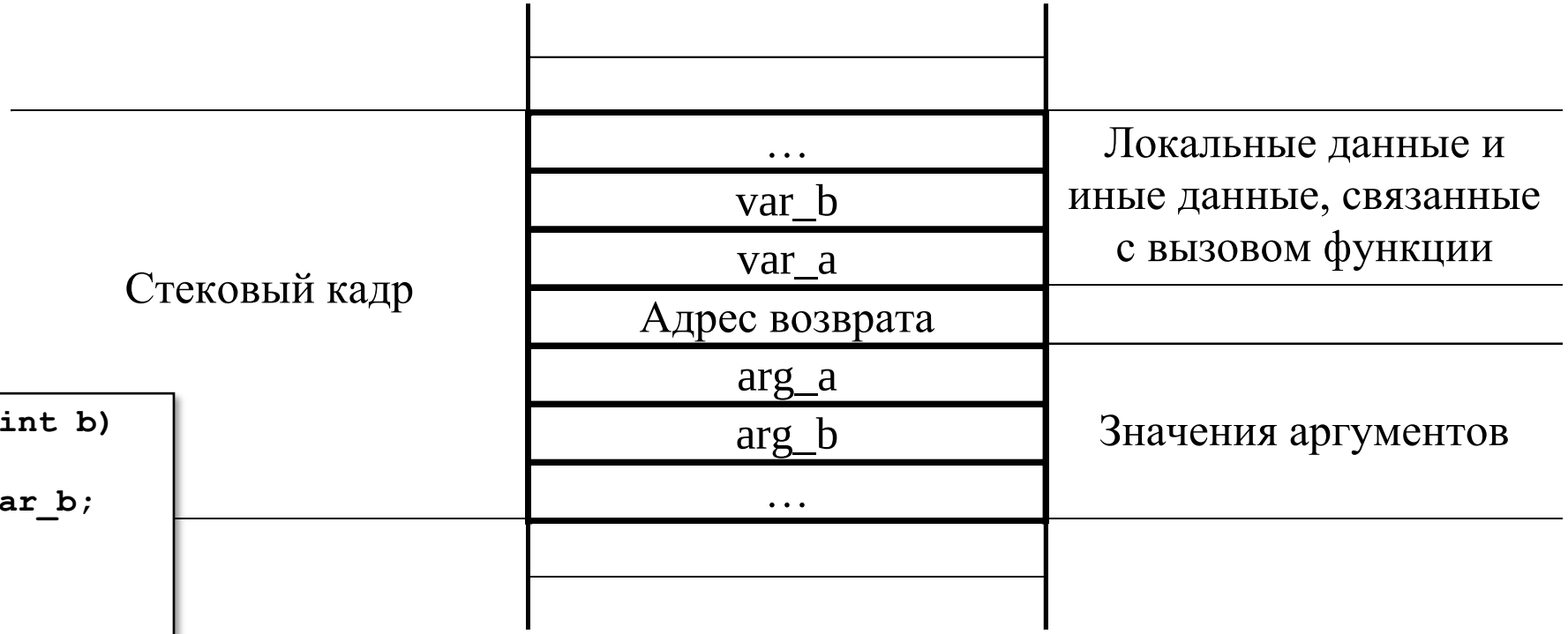
Стековый кадр

Стековый кадр (фрейм) - механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека.

В стековом кадре размещаются:

- значения фактических аргументов функции;
- адрес возврата;
- локальные переменные;
- иные данные, связанные с вызовом функции.

Стековый кад




```
void foo(int a, int b)
{
    int var_a, var_b;
    ...
}
...
{
    foo(arg_a, arg_b);
    // адрес возврата
```

Стековый кадр

“+”

- Удобство и простота использования.

“-”

- Производительность

  Передача данных через память без необходимости замедляет выполнение программы.
- Безопасность

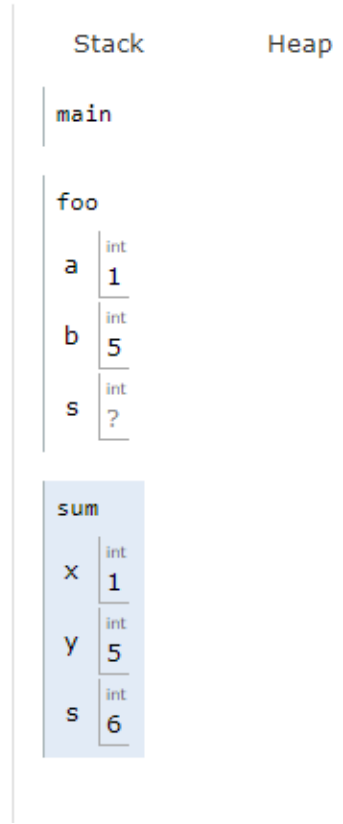
  Стековый кадр перемежает данные приложения с критическими данными - указателями, значениями регистров и адресами возврата.

Пример

```
int sum(int x, int y)
{
    int s = x + y;
    return s;
}
void foo(void)
{
    int a = 1, b = 5;
    int s = sum(a, b);
}
int main(void)
{
    foo();
    return 0;
}
```

C (gcc 4.8, C11)
(known limitations)

```
1  int sum(int x, int y)
2  {
3      int s = x + y;
4
5      return s;
6  }
7
8  void foo(void)
9  {
10     int a = 1, b = 5;
11     int s = sum(a, b);
12 }
13
14 int main(void)
15 {
16     foo();
17
18     return 0;
19 }
```



Пример (C vs asm)

```c
int sum(int x, int y)
{
    int s = x + y;

    return s;
}

void foo(void)
{
    int a = 1, b = 5;
    int s = sum(a, b);
}

int main(void)
{
    foo();

    return 0;
}
```

```
sum:
            push        ebp
            mov         ebp, esp
            sub         esp, 16
            mov         edx, DWORD PTR [ebp+12]
            mov         eax, DWORD PTR [ebp+8]
            add         eax, edx
            mov         DWORD PTR [ebp-4], eax
            mov         eax, DWORD PTR [ebp-4]
            leave
            ret

foo:
            push        ebp
            mov         ebp, esp
            sub         esp, 16
            mov         DWORD PTR [ebp-12], 1
            mov         DWORD PTR [ebp-8], 5
            push        DWORD PTR [ebp-8]
            push        DWORD PTR [ebp-12]
            call        sum
            add         esp, 8
            mov         DWORD PTR [ebp-4], eax
            leave
            ret
```

Пример (C vs asm)

Си	Ассемблер	Действия

`int a = 1, b = 5;`
`int s = sum(a, b);`
	`mov DWORD PTR [ebp-12], 1`
	`mov DWORD PTR [ebp-8], 5`
	`push DWORD PTR [ebp-8]`
	`push DWORD PTR [ebp-12]`
	`call sum`
	`add esp, 8`
	`mov DWORD PTR [ebp-4], eax`
		1. Помещение фактических параметров функции в стек (a – это [ebp-12], b – это [ebp-8]).
		2. Передача управления функции sum (в стек сохраняется адрес возврата – адрес инструкции add за call).

`int sum(int x, int y)`
`{`
	`push ebp`
	`mov ebp, esp`
	`sub esp, 16`
		1. Сохранение контекста вызвавшей функции.
		2. Формирование контекста функции sum.

`int s = x + y;`	`mov edx, DWORD PTR [ebp+12]`
	`mov eax, DWORD PTR [ebp+8]`
	`add eax, edx`
	`mov DWORD PTR [ebp-4], eax`
		1. Получение значений параметров (x это [ebp+8], y это [epb+12]).
		2. Вычисление суммы (add).
		3. Помещение результата в переменную s (s это [ebp-4]).

`return s;`
`}`
	`mov eax, DWORD PTR [ebp-4]`
	`leave`
	`ret`
		1. Формирование возвращаемого значения (mov).
		2. Восстановление контекста вызвавшей функции (leave = mov esp, ebp; pop ebp).
		3. Возврат управления (ret).

`int s = sum(a, b);`	`call sum`
	`add esp, 8`
	`mov DWORD PTR [ebp-4], eax`
		1. Очистка стека от параметров функции sum.
		2. Прием возвращаемого значения (s это [ebp-4]).

Ошибка: возвращение указателя на локальную переменную

```
#include <stdio.h>

char* make_greeting(const char *name)
{
    char str[64];

    snprintf(str, sizeof(str), "Hello, %s!", name);

    return str;
}

int main(void)
{
    char *msg = make_greeting("Petya");

    printf("%s\n", msg);

    return 0;
}
```

Ошибка: переполнение буфера

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char str[16];

    if (argc < 2)
        return 1;

    sprintf(str, "Hello, %s!", argv[1]);

    printf("%s (%d)\n", str, strlen(str));

    return 0;
}
```

Ошибка: указатель на переменную, вышедшую из области видимости

```
{
    int *p;

    {
        int b = 5;

        p = &b;

        printf("%d %d\n", *p, b);            // 5 5
    }

    printf("%d\n", b);          // ошибка компиляции
    printf("%d\n", *p);         // ошибка времени выполнения
}
```

Variable Length Array

```c
#include <stdio.h>

int main(void)
{
    int n;

    printf("n: ");
    scanf("%d", &n);

    int a[n];

    printf("n = %d, sizeof(a) = %d\n",
                    n, (int) sizeof(a));

    for (int i = 0; i < n; i++)
        a[i] = i;

    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

Variable Length Array

- Длина такого массива вычисляется во время выполнения программы, а не во время компиляции.
- Память под элементы массива выделяется на стеке.
- Массивы переменного размера нельзя инициализировать при определении.
- Массивы переменной длины могут быть многомерными.
- Адресная арифметика справедлива для массивов переменной длины.
- Массивы переменной длины облегчают описание заголовков функций, которые обрабатывают массивы.

alloca

```
#include <alloca.h>

void* alloca(size_t size);
```

Функция *alloca* выделяет область памяти, размером size байт, на стеке. Функция возвращает указатель на начало выделенной области. Эта область автоматически освобождается, когда функция, которая вызвала *alloca*, возвращает управления вызывающей стороне.

Если выделение вызывает переполнение стека, поведение программы не определено.

alloca

```
#include <alloca.h>
#include <stdio.h>

int main(void)
{
    int n;

    printf("n: ");
    scanf("%d", &n);

    int *a = alloca(n * sizeof(int));

    for (int i = 0; i < n; i++)
        a[i] = i;

    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

alloca

“+”

- Выделение происходит быстро.
- Выделенная область освобождается автоматически.

“–”

- Функция *нестандартная*.
- Серьезные ограничения по размеру области.
-

```
void foo(int size) {
    ...
    while(b){
        char tmp[size];
        ...
    }
```

```
void foo(int size) {
    ...
    while(b){
        char* tmp = alloca(size);
        ...
    }
```