Стек и куча.
Функции с переменным числом параметров.

Особенности использования локальных переменных

Для хранения локальных переменных используется так называемая автоматическая память.

“+”

- Память под локальные переменные выделяет и освобождает компилятор.

“-”

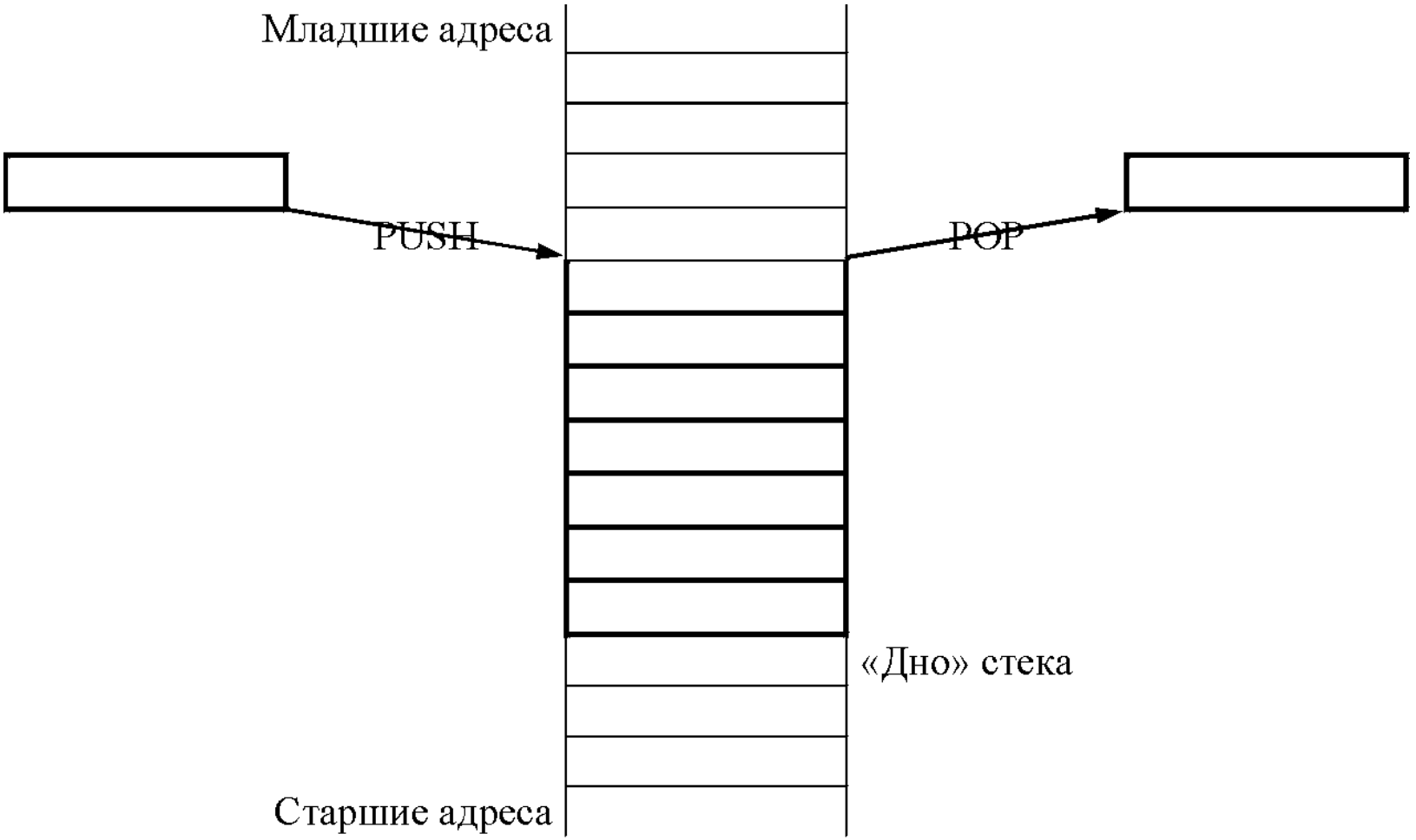
- Время жизни локальной переменной "ограничено" блоком, в котором она определена.
- Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции.
- Размер автоматической памяти в большинстве случаев ограничен.

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



Аппаратный стек используется для:

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

Аппаратный стек используется для:

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 …

Аппаратный стек используется для:

4. выделения и освобождения памяти под локальные переменные

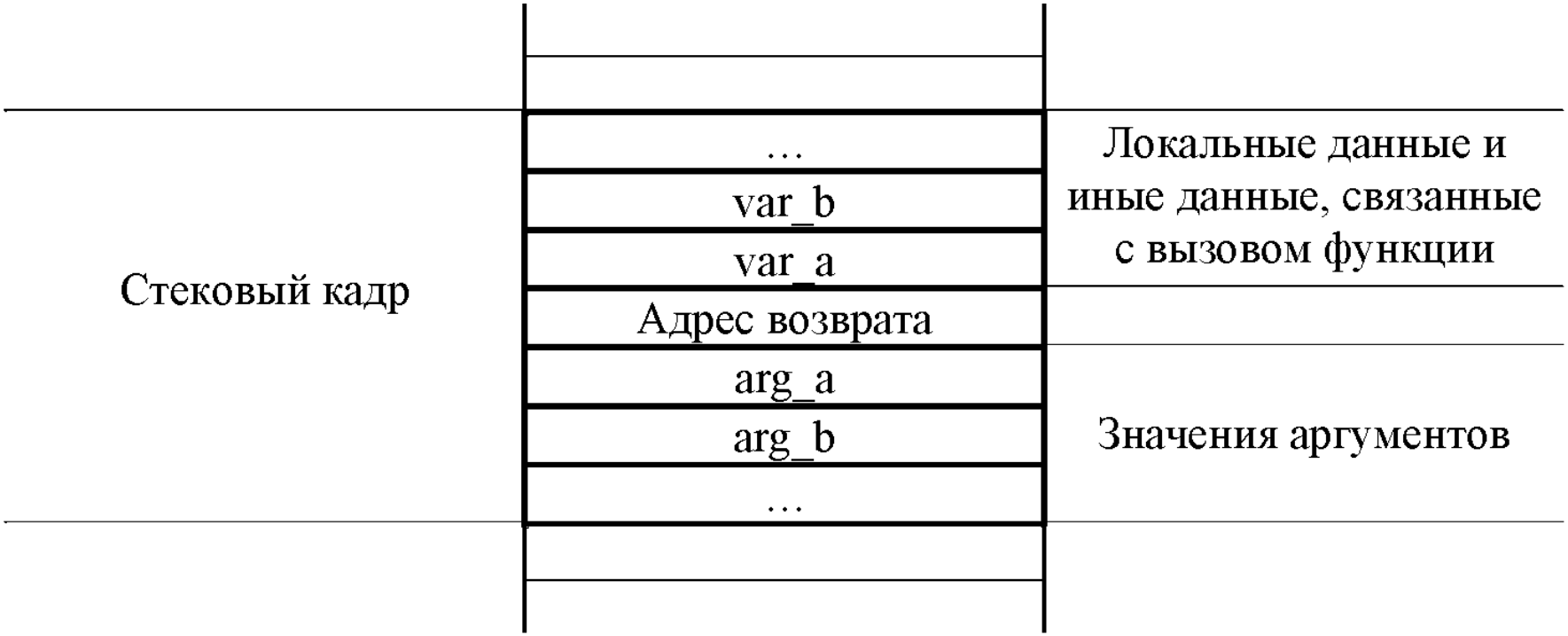
Стековый кадр (1)

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 кадр (2)



Стековый кадр (3)

“+”

- 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.

Передача параметров в функцию

```
#include <stdio.h>

void mul(int a)
{
    int b = 2;

    a = a * b;
}

int main(void)
{
    int n = 7;

    mul(n);

    printf("n = %d\n", n);

    return 0;
}
```

Посмотреть «демо» с помощью http://pythontutor.com/c.html#mode=edit

Адрес примера https://goo.gl/szezKi

Передача параметров в функцию

```
#include <stdio.h>

void mul(int *a)
{
    int b = 2;

    *a = (*a) * b;
}

int main(void)
{
    int n = 7;

    mul(&n);

    printf("n = %d\n", n);

    return 0;
}
```

Посмотреть «демо» с помощью http://pythontutor.com/c.html#mode=edit

Адрес примера https://goo.gl/BNKXQK

Рекурсия

```
#include <stdio.h>

int fact(int n)
{
    if (n == 0)
        return 1;
    else
    {
        int tmp = n * fact(n - 1);
        return tmp;
    }
}
int main(void)
{
    int n = 3;
    int f = fact(n);

    printf("fact(%d) = %d\n", n, f);

    return 0;
}
```

Посмотреть «демо» с помощью http://pythontutor.com/c.html#mode=edit

Адрес примера https://goo.gl/Hjn7xn

Пример (Си и ASM)

```
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 («за сценой»)

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]).

Особенности использования локальных переменных

```
{
    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
}
```

Посмотреть «демо» https://goo.gl/dEAZRH

Ошибка: возврат 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;
}
```

Последовательность действий при работе с динамической памятью (1).

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

int main(void)
{
    int *p = malloc(sizeof(int));
    if (!p)
        return -1;

    *p = 5;
    printf("%p %d\n", p, *p);   // адрес 5

    free(p);

    printf("%p\n", p);   // адрес (!)
    printf("%d\n", *p);  // ошибка времени выполнения

    return  0;
}
```

Последовательность действий при работе с динамической памятью (2).

- При запуске процесса ОС выделяет память для размещения кучи.
- Куча представляет собой непрерывную область памяти, поделённую на занятые и свободные области (блоки) различного размера.
- Информация о свободных и занятых областях кучи обычно храниться в списках различных форматов.

Последовательность действий при работе с динамической памятью (3).

Функция malloc выполняет примерно следующие действия:

- просматривает список занятых/свободных областей памяти, размещённых в куче, в поисках свободной области подходящего размера;
- если область имеет точно такой размер, как запрашивается, добавляет найденную область в список занятых областей и возвращает указатель на начало области памяти;
- если область имеет больший размер, она делится на части, одна из которых будет занята (выделена), а другая останется в списке свободных областей;

Последовательность действий при работе с динамической памятью (4).

«алгоритм» работы malloc (продолжение) :

- если область не удается найти, у ОС запрашивается очередной большой фрагмент памяти, который подключается к списку, и процесс поиска свободной области продолжается;
- если по тем или иным причинам выделить память не удалось, сообщает об ошибке (например, malloc возвращает NULL).

Последовательность действий при работе с динамической памятью (5).

Функция free выполняет примерно следующие действия:

- просматривает список занятых/свободных областей памяти, размещённых в куче, в поисках указанной области;
- удаляет из списка найденную область (или помечает область как свободную);
- если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то она сливается с ней в единую область большего размера.

Особенности использования динамической памяти

Для хранения данных используется «куча».

Создать переменную в «куче» нельзя, но можно выделить память под нее.

“+”

Все «минусы» локальных переменных.

“-”

Ручное управление временем жизни.

Функции с переменным числом параметров

```
int f(...);
```

- Во время компиляции компилятору не известны ни количество параметров, ни их типы.
- Во время компиляции компилятор не выполняет никаких проверок.

НО список параметров функции с переменным числом аргументов совсем пустым быть не может.

```
int f(int k, ...);
```

Функции с переменным числом параметров

Напишем функцию, вычисляющую среднее арифметическое своих аргументов.

Проблемы:

1. Как определить адрес параметров в стеке?
2. Как перебирать параметры?
3. Как закончить перебор?

Функции с переменным числом параметров

```
#include <stdio.h>

double avg(int n, ...)
{
    int *p_i = &n;
    double *p_d =
                (double*) (p_i+1);
    double sum = 0.0;

    if (!n)
        return 0;

    for (int i = 0; i < n;
                        i++, p_d++)
        sum += *p_d;

    return sum / n;
}
```

```
int main(void)
{
    double a =
      avg(4, 1.0, 2.0, 3.0, 4.0);

    printf("a = %5.2f\n", a);

    return 0;
}
```

Функции с переменным числом параметров

```
double avg(double a, ...)
{
    int n = 0;
    double *p_d = &a;
    double sum = 0.0;

    while (*p_d)
    {
        sum += *p_d;
        n++;

        p_d++;
    }

    if (!n)
        return 0;

    return sum / n;
}
```

```
int main(void)
{
    double a =
         avg(1.0, 2.0, 3.0,
                        4.0, 0.0);

    printf("a = %5.2f\n", a);

    return 0;
}
```

Функции с переменным числом параметров

```
#include <stdio.h>

void print_ch(int n, ...)
{
    int *p_i = &n;
    char *p_c = (char*) (p_i+1);

    for (int i = 0; i < n; i++, p_c++)
        printf("%c %d\n", *p_c, (int) *p_c);
}

int main(void)
{
    print_ch(5, 'a', 'b', 'c', 'd', 'e');

    return 0;
}
```

Стандартный способ работы с параметрами функций с переменным числом параметров

stdarg.h

- va_list
- void va_start(va_list argptr, last_param)
- type va_arg(va_list argptr, type)
- void va_end(va_list argptr)

Функции с переменным числом параметров

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

double avg(int n, ...)
{
    va_list vl;
    double num;
    double sum = 0.0;

    if (!n)
        return 0;

    va_start(vl, n);

    for (int i = 0; i < n; i++)
    {
        num = va_arg(vl, double);
        sum += num;
    }

    va_end(vl);

    return sum / n;
}
```

```
int main(void)
{
    double a =
      avg(4, 1.0, 2.0, 3.0, 4.0);

    printf("a = %5.2f\n", a);

    return 0;
}
```

Функции с переменным числом параметров

```
#include <stdarg.h>
#include <stdio.h>
double avg(double a, ...)
{
    va_list vl;
    int n = 0;
    double num, sum = 0.0;

    va_start(vl, a);
    num = a;

    while (num)
    {
        sum += num;
        n++;
        num = va_arg(vl, double);
    }

    va_end(vl);

    if(!n)
        return 0;
    return sum / n;
}
```

```
int main(void)
{
    double a =
         avg(1.0, 2.0, 3.0,
                        4.0, 0.0);

    printf("a = %5.2f\n", a);

    return 0;
}
```

Функции с переменным числом параметров: журналирование

```
// log.c

#include <stdio.h>

static FILE* flog;

int log_init(const char
                        *name)
{
    flog = fopen(name, "w");
    if(!flog)
        return 1;

    return 0;
}

FILE* log_get(void)
{
    return flog;
}

void log_close(void)
{
    fclose(flog);
}
```

```
// log.h

#ifndef __LOG__H__

#define __LOG__H__

#include <stdio.h>

int log_init(const char
                    *name);

FILE* log_get(void);

void log_close(void);

#endif  //  __LOG__H__
```

```
// log.c
#include <stdio.h>
#include <stdarg.h>

static FILE* flog;

int log_init(const char
                        *name)
{
    flog = fopen(name, "w");
    if(!flog)
        return 1;

    return 0;
}

void log_message(const char
                  *format, ...)
{
    va_list args;
    va_start(args, format);
    vfprintf(flog, format, args);
    va_end(args);
}

void log_close(void)
{
    fclose(flog);
}
```

```
// log.h

#ifndef __LOG__H__

#define __LOG__H__

#include <stdio.h>

int log_init(const char
                   *name);

void log_message(const char
                  *format, ...);

void log_close(void);

#endif  //  __LOG__H__
```

Литература

1. Б. Керниган, Д. Ритчи «Язык программирования Си» (глава 8 «Интерфейс системы Unix», подраздел «Пример распределения памяти»)
2. С. Прата «Язык программирования Си» (глава 16 «Переменные аргументы: файл stdarg.h»)
3. Черновик стандарта C99

36