СибГУТИ
Кафедра вычислительных систем

ПРОГРАММИРОВАНИЕ

Время жизни и области видимости программных объектов

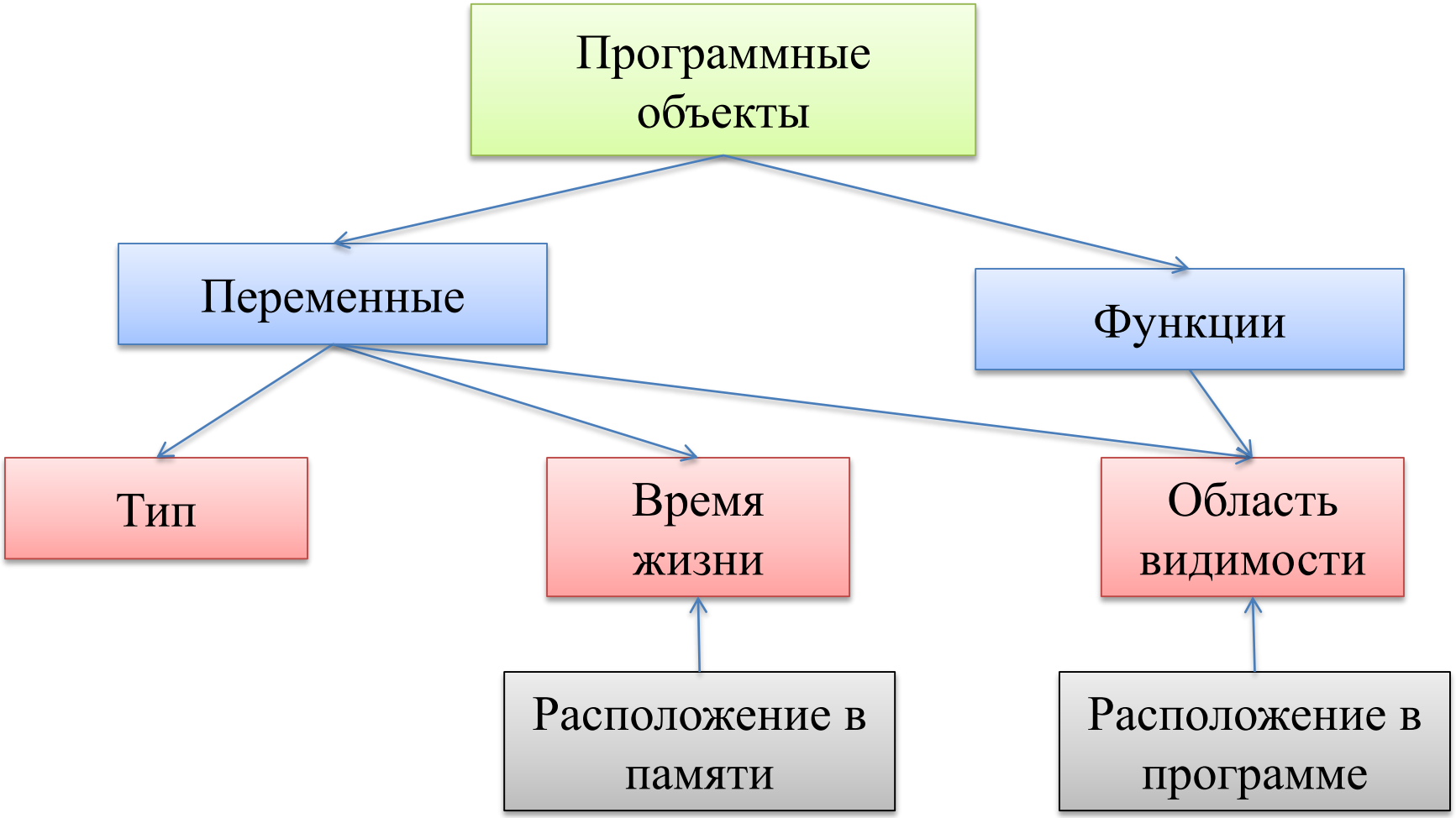
Преподаватель:
ст. преп. Кафедры ВС **Перышкова Е.Н.**

Рассматриваемые вопросы

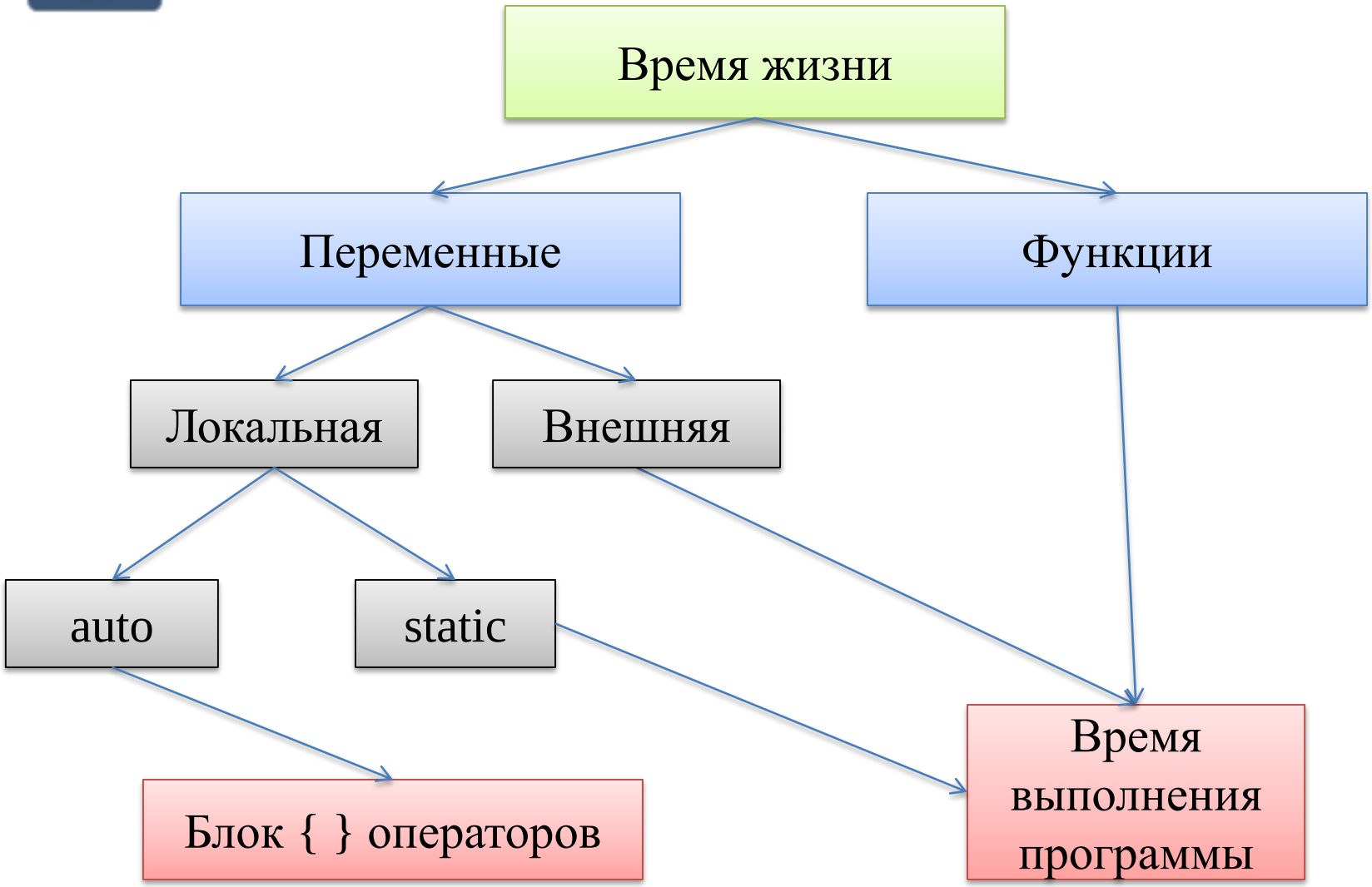
1. **Виды программных объектов и их характеристики.**
2. **Организация памяти программы:**
   - **сегмент кода;**
   - **сегмент данных;**
   - **регионы динамических библиотек;**
   - **стек вызовов и его функционирование;**
   - **динамическая память.**
3. **Классы памяти программных объектов:**
   - **классы памяти переменных;**
   - **классы памяти функций.**

Виды программных объектов и их характеристики

Классификация и характеристики программных объектов



Время жизни


Время жизни
Переменные
Функции
Локальная
Внешняя
auto
static
Блок { } операторов
Время выполнения программы

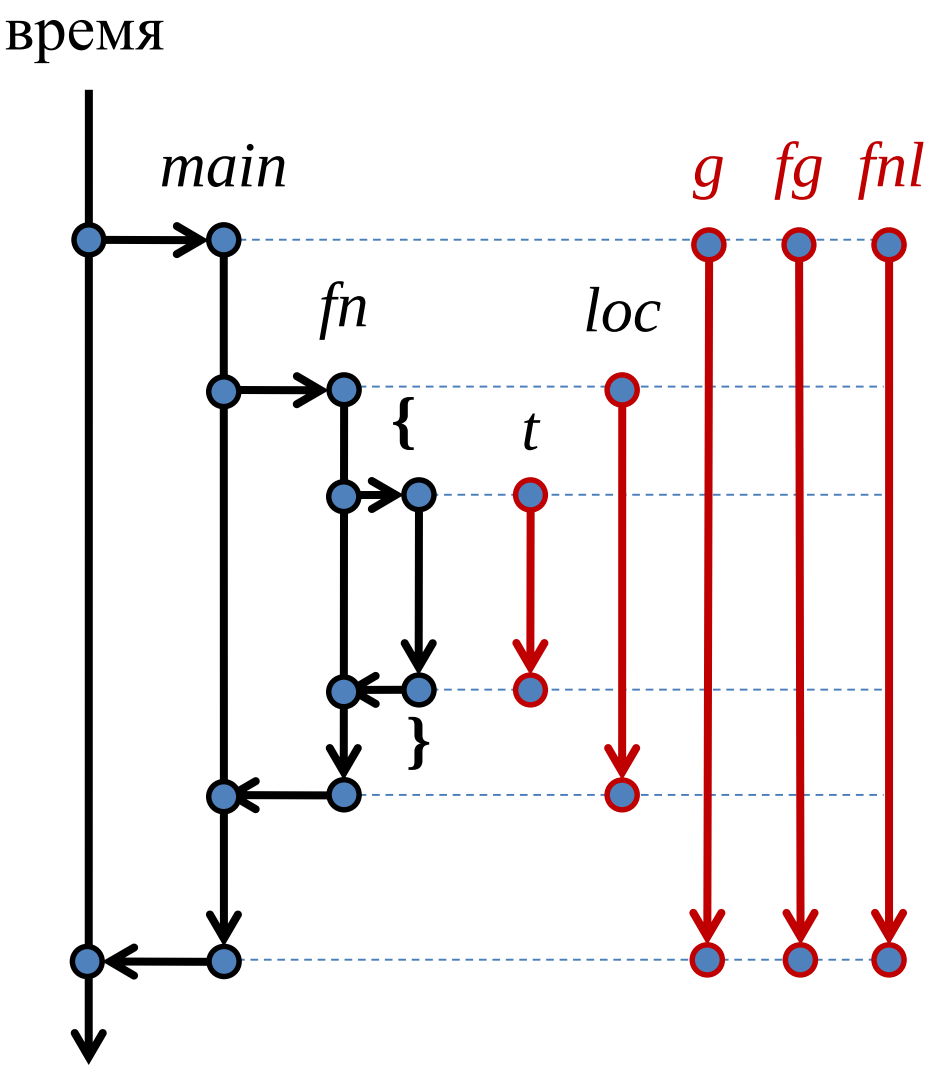
Переменные: внешняя vs локальная

main.c

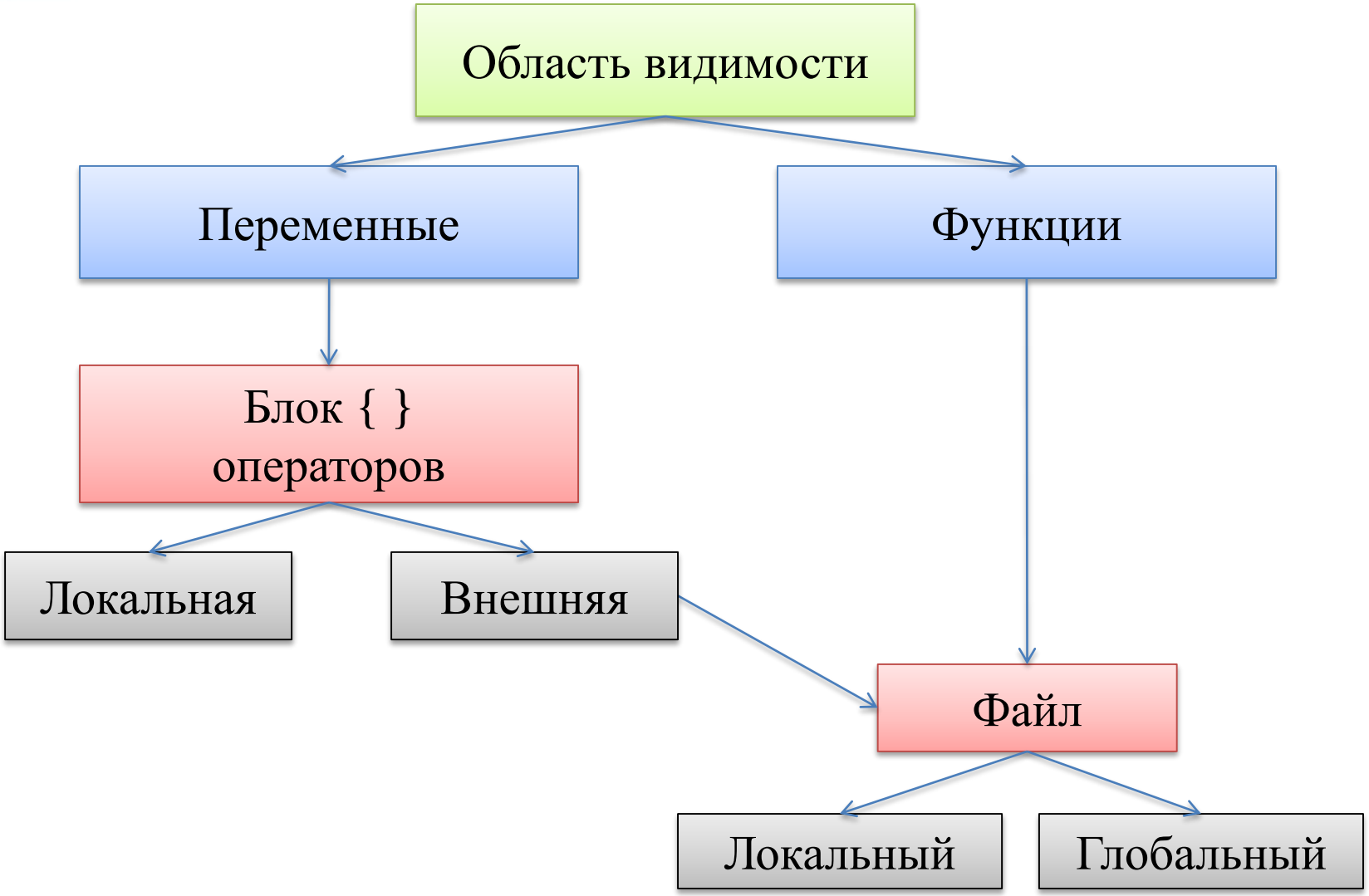
```
int g = 0;
static int fg = 0;

static int fn( ){
    int loc;
    static int fnl;
    {
        int t = 10;
        loc = t + 1;
    }
    fnl = loc*10;
}

int main(){
    ...
    fn();
    ...
}
```



Расположение объектов в программе



Область видимости

func.c

```
int global = 0;
static int file_local = 0;
static int f1( ){
  int x = file_local;
  static int y;
  y++;
}
int f2( ){
  int y;
  file_local += global;
}
```

main.c

```
int f2( );
extern int global;

int main( )
{
  global = 5;
  f2();
}
```

func.o

```
export: global, f2
need:
local: file_local, f1
```

main.o

```
export: main
need: f2, global
local:
```

Область видимости
(конфликт глобальных объектов)

func.c

```
int global = 0;
static int file_local = 0;
static int f1( ){
  int x = file_local;
  static int y;
  y++;
}
int f2( ){
  int y;
  file_local += global;
}
```

main.c

```
int f2( );
extern int global;
static int file_local = 10;

int main( )
{
  global = 5;
  f2();
}
static int f1( ){ ... }
```

func.o

```
export: global, f2 , f1
need:
local: file_local
```

main.o

```
export: main, f1
need: f2, global
local: file_local
```

Область видимости
(перекрытие глобальных объектов статическими)

func.c

```
#include <stdio.h>

int test = 10;
int f(){
    printf("f: "
       "test = %d\n",test);
}
```

main.c

```
#include <stdio.h>

static int test = 5;
int main()
{
    printf("main: "
       "test = %d\n",test);
    f();
}
```

func.o

```
export: test, f
need:
local:
```

main.o

```
export: main
need: f
local: test
```

Область видимости
(перекрытие глобальных объектов статическими, 2)

func.c

```
#include <stdio.h>

int test = 10;
int f(){
    printf("f: "
       "test = %d\n",test);
}
```

main.c

```
#include <stdio.h>

static int test = 5;
int main()
{
    printf("main: "
       "test = %d\n",test);
    f();
}
```

func.o

```
export: test, f
need:
local:
```

n.o

```
$ ./demo_prog_1
main: test = 5
f: test = 10
```

Область видимости
(перекрытие глобальных объектов локальными)

main.c

```
#include <stdio.h>
int v1 = 5;
static int v2 = 5;

int main()
{
    float v1 = 1, v2 = 2;
    ...
    {
        int v2 = 10;
    }
    ...
}
```

Объекты с меньшей областью видимости **перекрывают** объекты с более широкой областью видимости.

Области видимости могут только **вкладываться** друг в друга.

Область видимости
(дублирование имен локальных объектов)

func.c

```
#include <stdio.h>

void f(int L2, int x)
{
    int L1 = L2 + x;
    printf("f: L1=%d,"
       "L2=%d\n",L1, L2);
}
```

main.c

```
#include <stdio.h>

int main()
{
    int L1 = 10, L2 = 20;
    f(L1, L2);
    printf("main: L1=%d,"
       L2=%d\n",L1, L2);
    return 0;
}
```

func.o

```
export: f
need:
local:
```

main.o

```
export: main
need: f
local:
```

Область видимости
(дублирование имен локальных объектов, 2)

func.c

```
#include <stdio.h>

void f(int L2, int x)
{
    int L1 = L2 + x;
    printf("f: L1=%d,"
       "L2=%d\n",L1, L2);
}
```

main.c

```
#include <stdio.h>

int main()
{
    int L1 = 10, L2 = 20;
    f(L1, L2);
    printf("main: L1=%d,"
       L2=%d\n",L1, L2);
    return 0;
}
```

```
$ ./demo_prog_2
f: L1=30, L2=10
main: L1=10, L2=20
```

func.

```
export: f
need:
local:
```

ain.o

```
n
local:
```

Организация памяти программы

Регионы памяти программы

```
#include <stdio.h>
int main(){
        sleep(100);
}
```

demo.c

```
$ ./demo &
$ cat /proc/`pgrep demo`/maps
00400000-00401000 r-xp                    demo
00600000-00601000 r--p                    demo
00601000-00602000 rw-p                    demo
00b42000-00b63000 rw-p                    [heap]
7fa007e53000-7fa008012000 r-xp            /lib/libc-2.17.so
...
7fa008215000-7fa008217000 rw-p            /lib/libc-2.17.so
7fa008217000-7fa00821c000 rw-p
7fa00821c000-7fa00823f000 r-xp            /lib/ld-2.17.so
7fa00841a000-7fa00841d000 rw-p
7fa00843c000-7fa00843e000 rw-p
7fa00843e000-7fa00843f000 r--p            /lib/ld-2.17.so
7fa00843f000-7fa008441000 rw-p            /lib/ld-2.17.so
7fff1966d000-7fff1968e000 rw-p            [stack]
7fff1970b000-7fff1970d000 r-xp            [vdso]
ffffffffff600000-ffffffffff601000 r-xp    [vsyscall]
```

Организация памяти программы

Права: Read (R), Write (W), eXecute (X)

0x00400000 – 0x00401000, размер: $1000_{16} = 4096_{10}$

0x00600000 – 0x00601000, размер: $1000_{16} = 4096_{10}$

0x00601000 – 0x00602000, размер: $1000_{16} = 4096_{10}$

0x00b42000 – 0x00b63000, размер: $21000_{16} = 135168_{10}$
33 страницы памяти

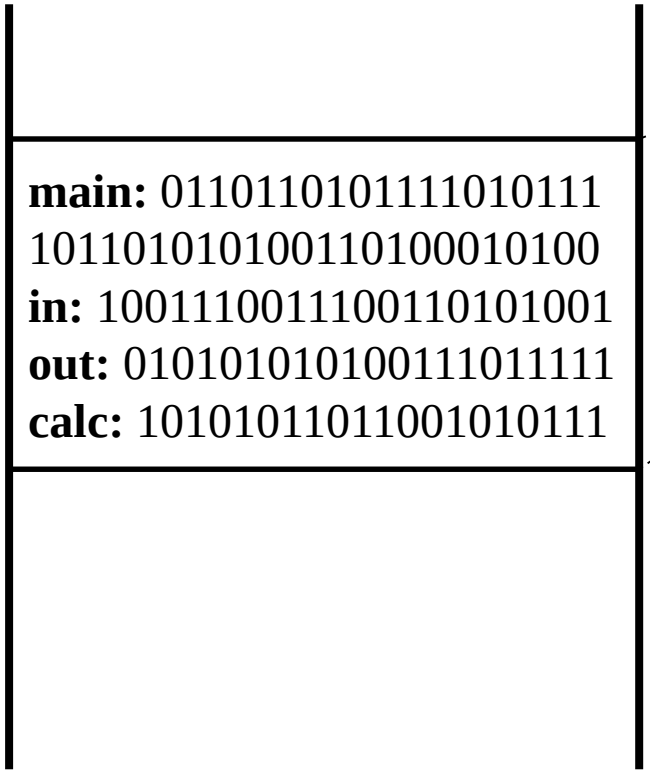
0x7fa007e53000 – 0x7fa008441000,
размер: $5EE000_{16} = 6217728_{10}$
1518 страниц памяти

7fff1966d000-7fff1968e000,
размер: $21000_{16} = 135168_{10}$
33 страниц памяти

Сегмент кода

Содержимое сегмента кода загружается из исполняемого файла, расположенного на носителе информации и содержит машинный код, сформированный компилятором.

Доступ: *чтение и исполнение*.

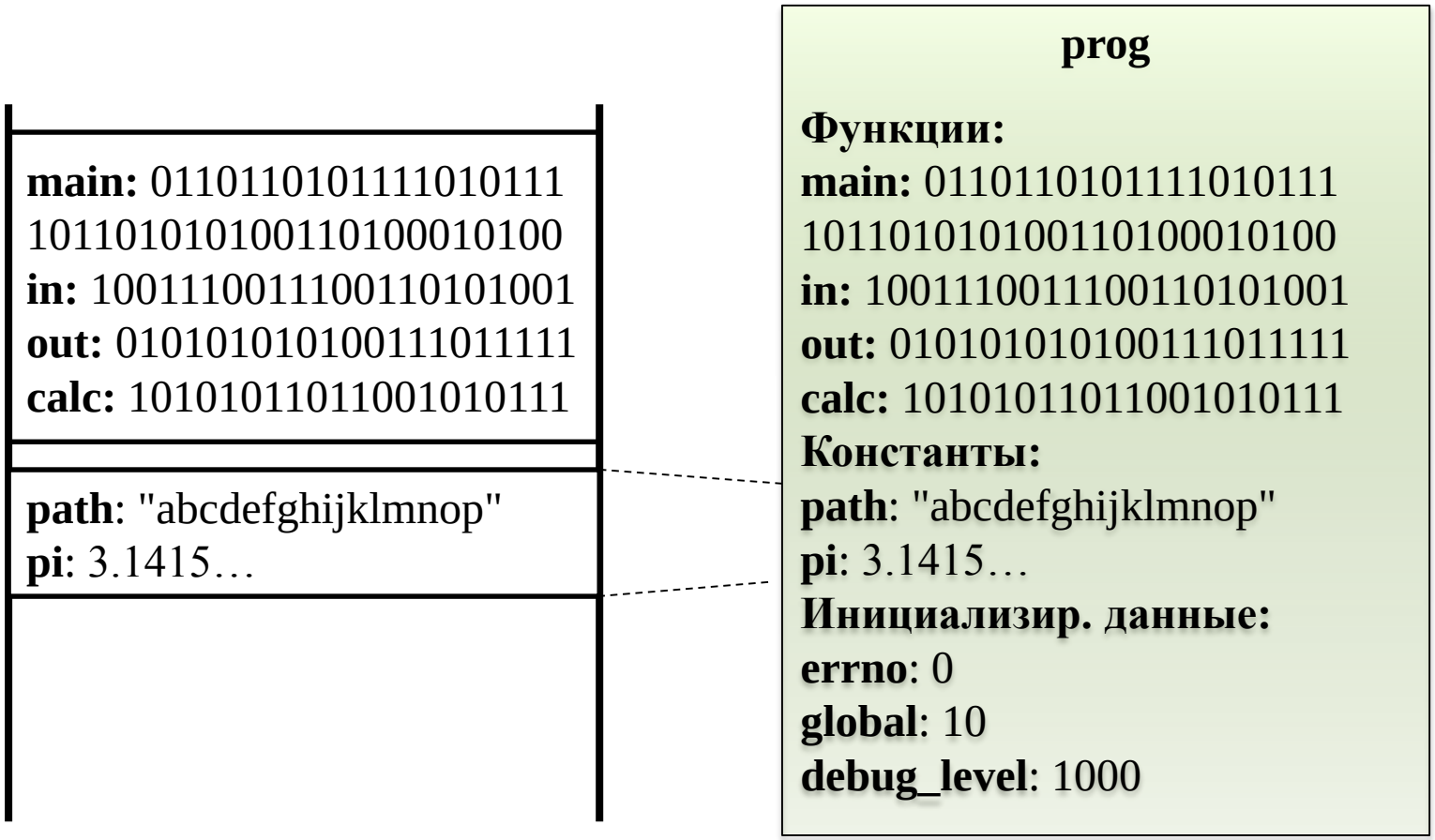


prog

Функции:
main: 01101101011110101111010110110101010011010001010
in: 10011100111001101010010101000111011111
out: 01010101010011101111
calc: 1010101101100101011
Константы:
path: "abcdefghijklmnop"
pi: 3.1415…
Инициализир. данные:
errno: 0
global: 10
debug_level: 1000

Сегмент данных (констант)

Объявленные в программе константы размещаются в специальном регионе памяти, к которому возможен доступ *только чтение*.

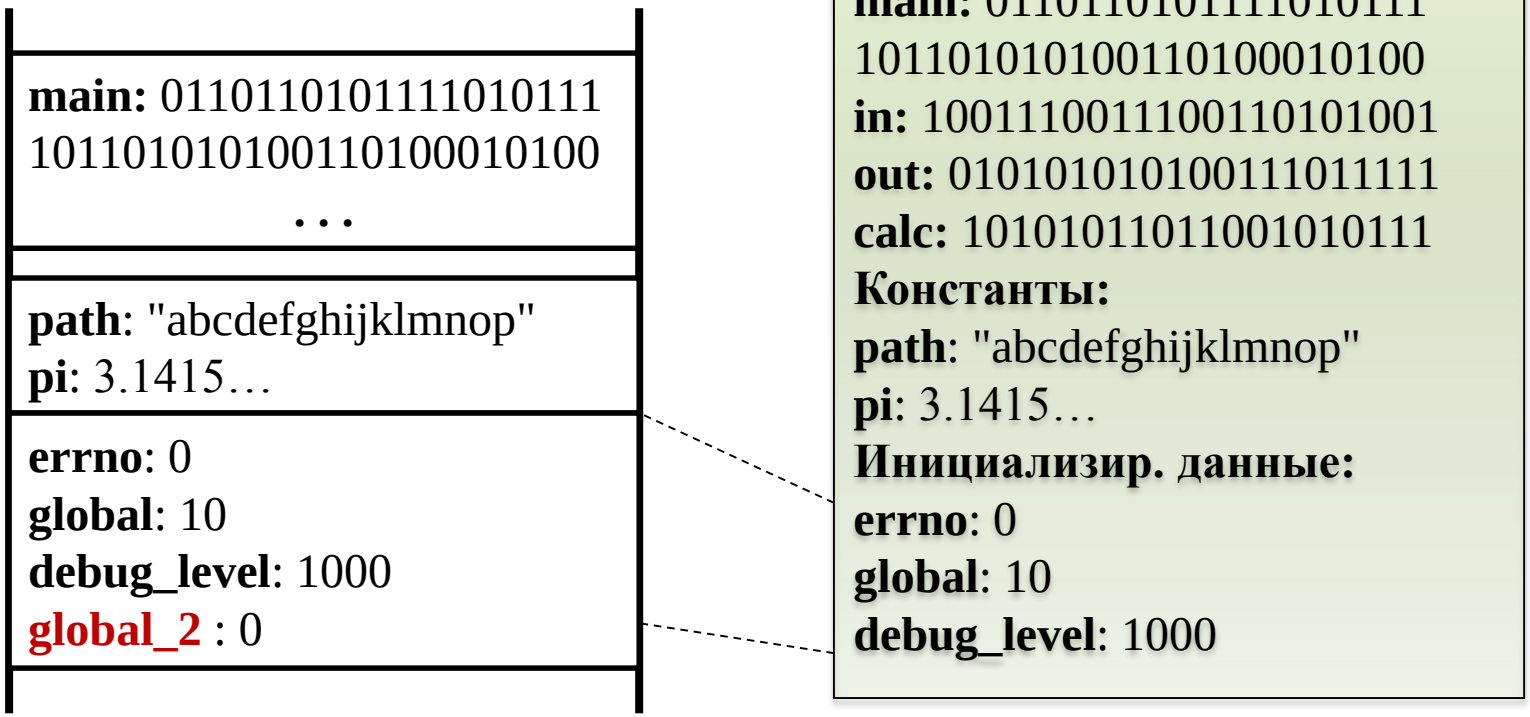


Сегмент данных (модифицируемые)

Сегмент, содержащий статические и глобальные переменные, которые существуют на протяжении всего времени выполнения программы размещаются в отдельном сегменте.

Инициализирующие значения копируются из исполняемого файла.

Доступ: *чтение и запись*.



Регионы динамических библиотек

- Динамические библиотеки содержат как код так и данные, их загрузка осуществляется аналогично сегменту кода и данных соответственно.
- Дополнительной функцией данных сегментов является взаимная защита данных кучи от данных стека, которые растут навстречу друг другу. Это достигается ха счет регионов, для которых запрещена запись.

```
00b42000-00b63000 rw-p          [heap]
7f4d38afe000-7f4d38cbd000 r-xp /lib/libc-2.17.so
7f4d38cbd000-7f4d38ebc000 ---p /lib/libc-2.17.so
7f4d38ebc000-7f4d38ec0000 r--p /lib/libc-2.17.so
7f4d38ec0000-7f4d38ec2000 rw-p /lib/libc-2.17.so
7f4d38ec2000-7f4d38ec7000 rw-p
7f4d38ec7000-7f4d38eea000 r-xp /lib/ld-2.17.so
7f4d390c5000-7f4d390c8000 rw-p
7f4d390e7000-7f4d390e9000 rw-p
7f4d390e9000-7f4d390ea000 r--p /lib/ld-2.17.so
7f4d390ea000-7f4d390ec000 rw-p /lib/ld-2.17.so
7fff18da9000-7fff18dca000 rw-p [stack]
```

Динамическое размещение программных объектов памяти

В процессе функционирования программы в ряде случаев возникает необходимость выделения памяти, которая не может быть предусмотрена заранее на этапе компиляции (статически), в частности:

1. Компилятор не может предугадать сколько раз и в какие моменты времени будет вызвана та или иная функция и будет ли она вызвана вообще. С другой стороны каждая функция имеет локальные переменные, которые необходимо где-то размещать. При этом после завершения функции эта область данных больше не нужна и может быть использована для других целей. Для обеспечения указанного функционала в программе используется *стек вызовов*.

2. При реализации многих программных архитектур возникает необходимость выделения памяти, не зависящей от той или иной функции. Время жизни такой памяти должно определяться явно программистом, а не тем, какие функции вызваны в данный момент. Такая память называется *динамической* и выделяется в области динамической памяти (куче, англ. heap).

Сегмент стека (стек)

Сегмент стека (call stack – стек вызовов) в теории вычислительных систем, LIFO-стек, хранящий информацию для:

1) возврата управления из вызванной подпрограммы в вызывающую подпрограмму;

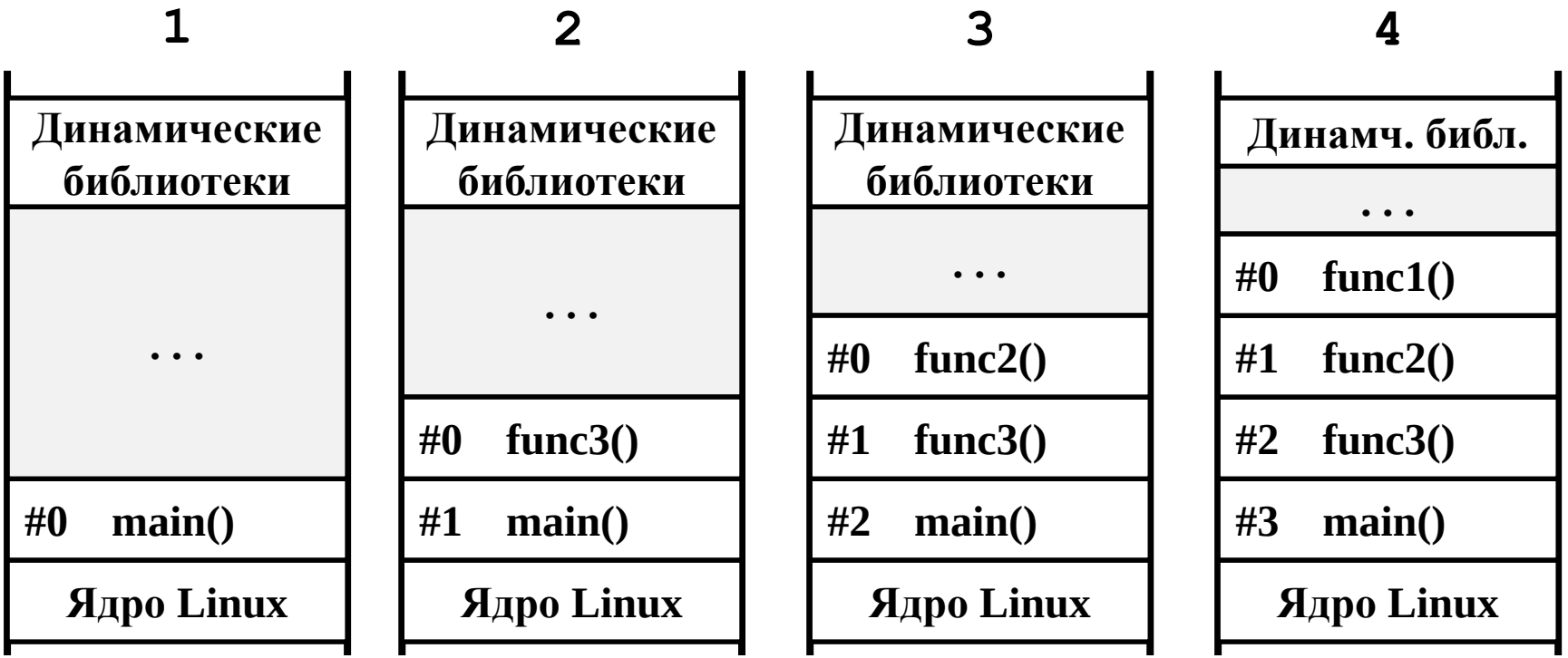
2) возврата в подпрограмму из обработчика прерывания (в том числе при переключении задач в многозадачной среде).

Стек *также используется для хранения локальных переменных* каждой вызванной функции.

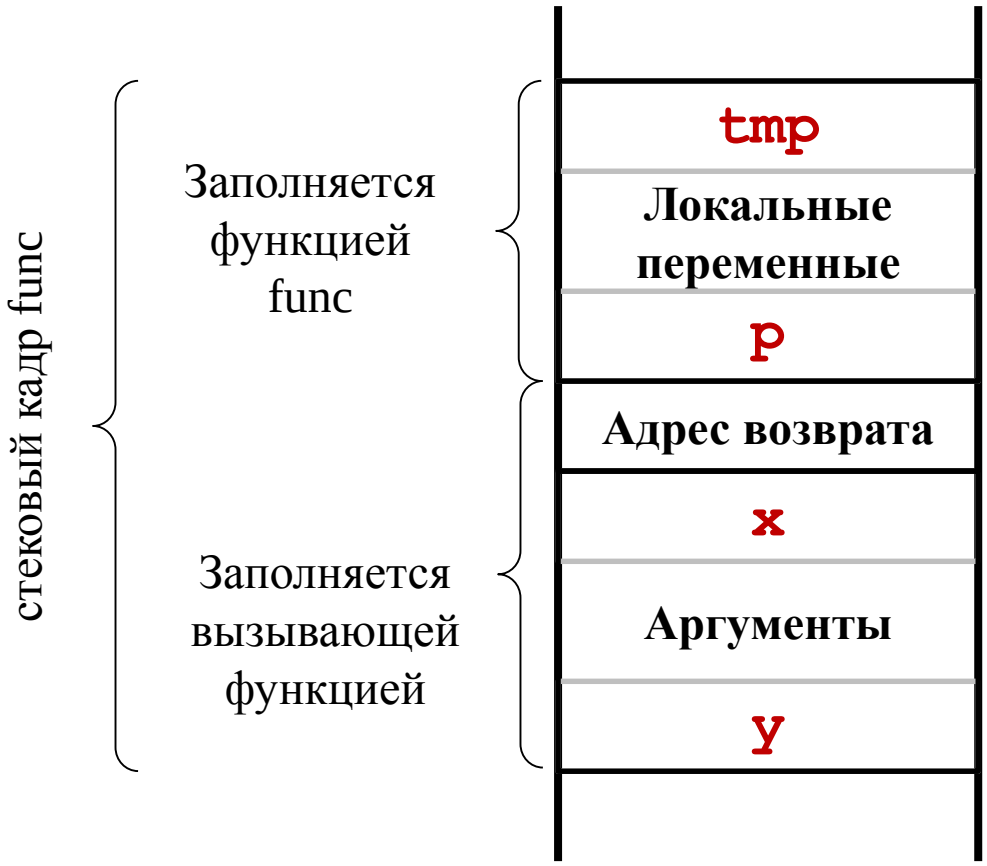
Для каждой вызываемой функции в стеке выделяется область памяти, называемая кадром (frame)

Пример работы стека

```
4 int func1(...){ ... }
3 int func2(...){ ... func1(...) ...}
2 int func3(...){ ... func2(...) ...}
1 int main(){ ... func3(...) ... }
```



Стековый кад

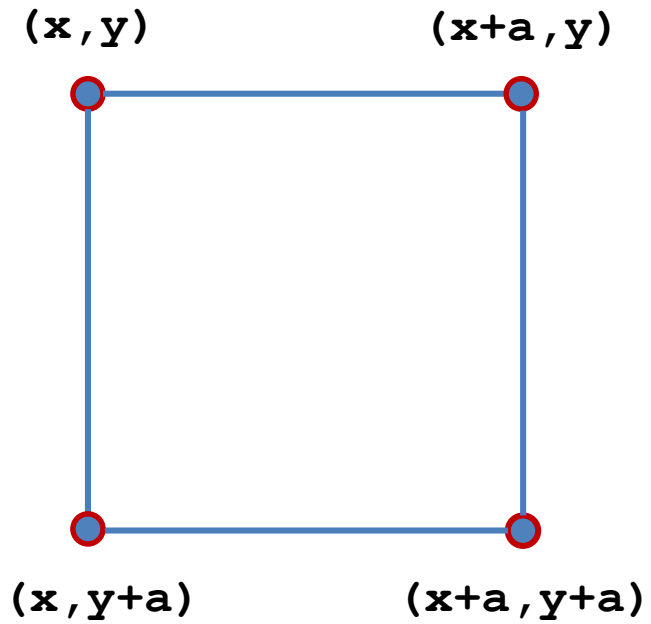



```
int func(int x, float y)
{
     int tmp = x + y;
     char p[16];
     sprintf(p,"%d",tmp);
     return 10*tmp;
}
```

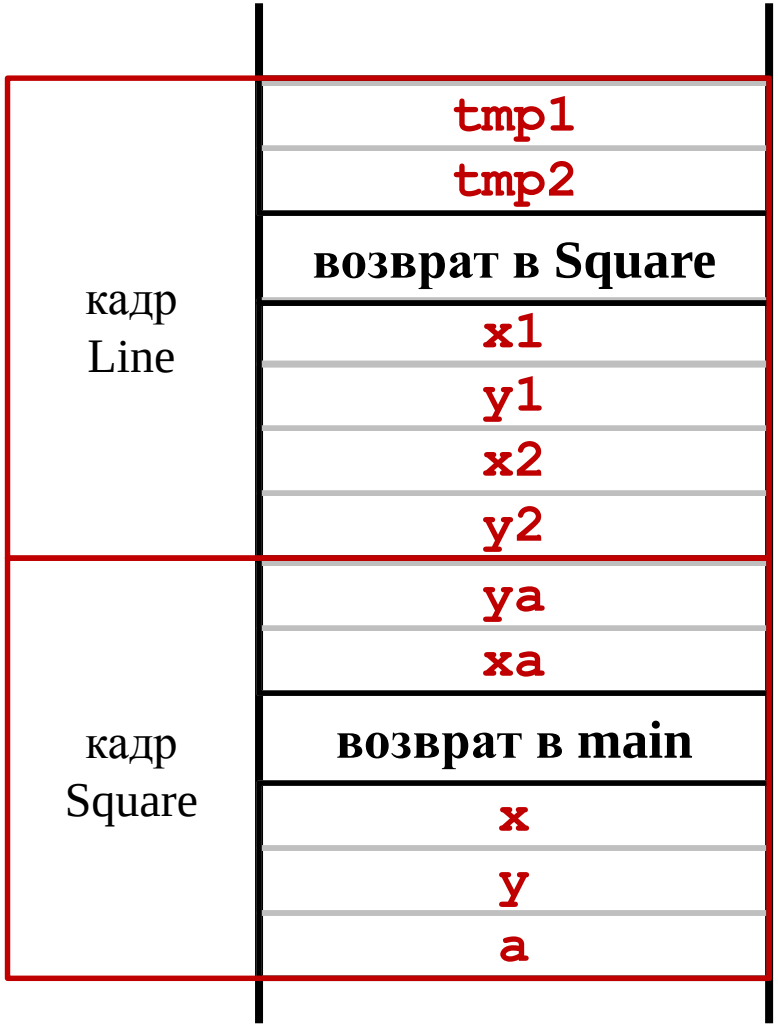
Стековый кадр (2)

```
int Line(int x1, int y1,
         int x2, int y2)
{
     // do something to
     // draw a line
}

int Square(int x, int y,
           int a)
{
     int xa = x + a;
     int ya = y + a;
     Line(x,y,xa,y);
     Line(x,y,x,ya);
     Line(xa,y,xa,ya);
     Line(x,ya,xa,ya);
}
```



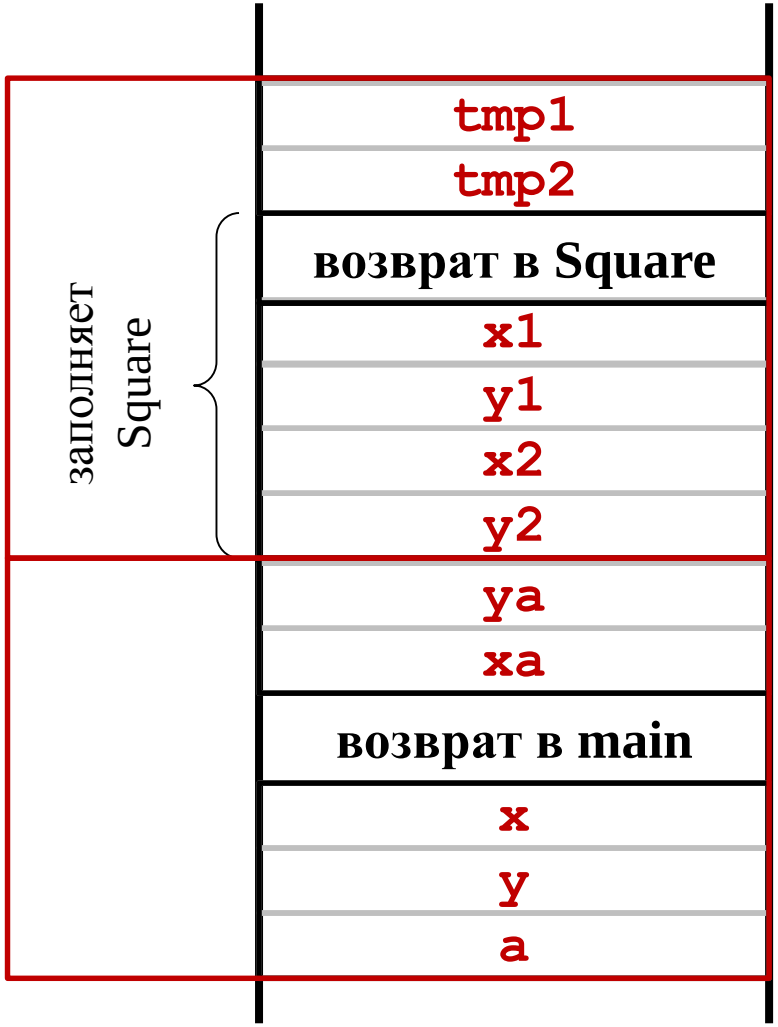
Стековый кадр (2)




```
int Line(int x1, int y1,
         int x2, int y2)
{
     int tmp1, tmp2;
     // do something to
     // draw a line
}

int Square(int x, int y,
           int a)
{
     int xa = x + a;
     int ya = y + a;
     Line(x,y,xa,y);
     Line(x,y,x,ya);
     Line(xa,y,xa,ya);
     Line(x,ya,xa,ya);
}
```

Стековый кадр (2)




```
int Line(int x1, int y1,
         int x2, int y2)
{
     int tmp1, tmp2;
     // do something to
     // draw a line
}

int Square(int x, int y,
           int a)
{
     int xa = x + a;
     int ya = y + a;
     Line(x,y,xa,y);
     Line(x,y,x,ya);
     Line(xa,y,xa,ya);
     Line(x,ya,xa,ya);
}
```

Анализ стека с GDB

Отладчик GNU GDB предоставляет следующие инструменты работы со стеком:

- просмотр содержимого стека;
- переход между кадрами стека (смена контекста).

Первая операция выполняется командой backtrace:

значения фактических параметров

```
(gdb) backtrace
#0  Line (x1=10, y1=10, x2=15, y2=10) at test.c:6
#1  0x000000000040059c in Square (x=10, y=10, a=5) at test.c:13
#2  0x00000000004005ef in main () at test.c:21
```

номера стековых кадров

адрес возврата

имя функции

расположение в исходном коде

Анализ стека с GDB (2)

Отладчик GNU GDB предоставляет следующие инструменты работы со стеком:

- просмотр содержимого стека;
- переход между кадрами стека (смена контекста).

Вторая операция выполняется командой frame:

```
(gdb) backtrace
#0  Line (x1=10, y1=10, x2=15, y2=10) at test.c:6
#1  0x000000000040059c in Square (x=10, y=10, a=5) at test.c:13
#2  0x00000000004005ef in main () at test.c:21
(gdb) frame 1
#1  0x000000000040059c in Square (x=10, y=10, a=5) at test.c:13
13   Line(x,y,xa,y);
14   (gdb) p xa
$1 = 15
```

Анализ стека с GDB [пример]

Рассмотрим пример применения инструментов анализа стека GDB для обнаружения ошибки при вызове функции scanf. Вместо указателя на ячейку-получатель в scanf передается ее значение (*i* = 0).

```
#include <stdio.h>

int main()
{
    int i = 0;
    scanf("%d",i);
    printf("i = %d\n", i);
    return 0;
}
```

Анализ стека с GDB [пример] (2)

```
$ gdb bad_ptr
. . .
(gdb) r
Starting program: bad_ptr
10
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7a6817d in _IO_vfscanf () from libc.so.6
(gdb) bt
#0  0x00007ffff7a6817d in _IO_vfscanf () from libc.so.6
#1  0x00007ffff7a6fa22 in __isoc99_scanf () from libc.so.6
#2  0x00000000004005bf in main () at bad_ptr.c:6
(gdb) frame 2
#2  0x00000000004005bf in main () at bad_ptr.c:6
6           scanf("%d",i);
(gdb) p i
$1 = 0
```

Рекурсия

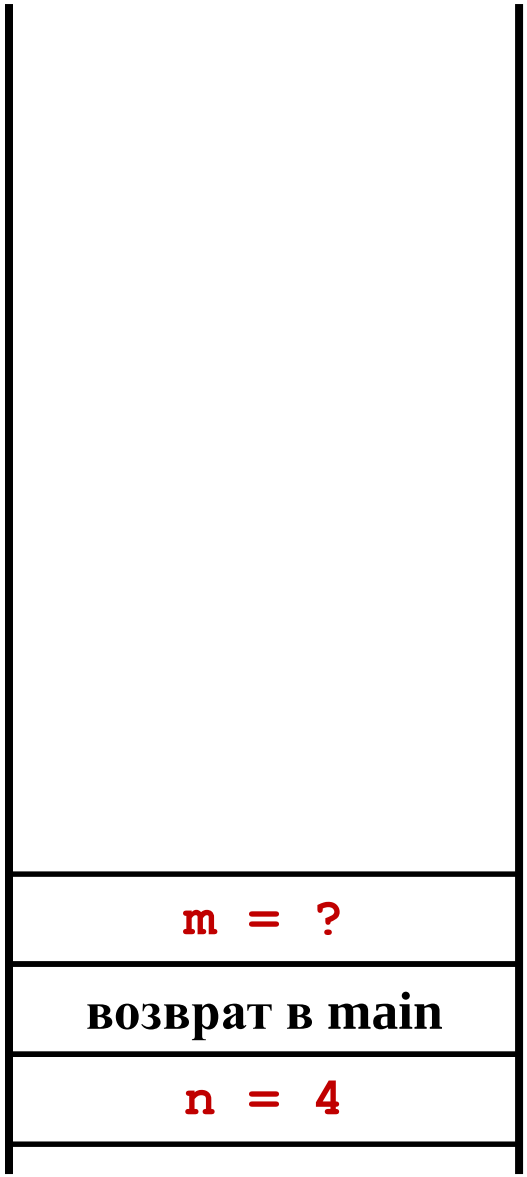
Наличие стека позволяет организовывать рекурсивные вызовы функций. В качестве примера рассмотрим задачу вычисления факториала:

$$n! = n \cdot (n - 1)!$$

```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}
```

Стек позволяет разграничить локальные переменные различных экземпляров одной и той же функции.

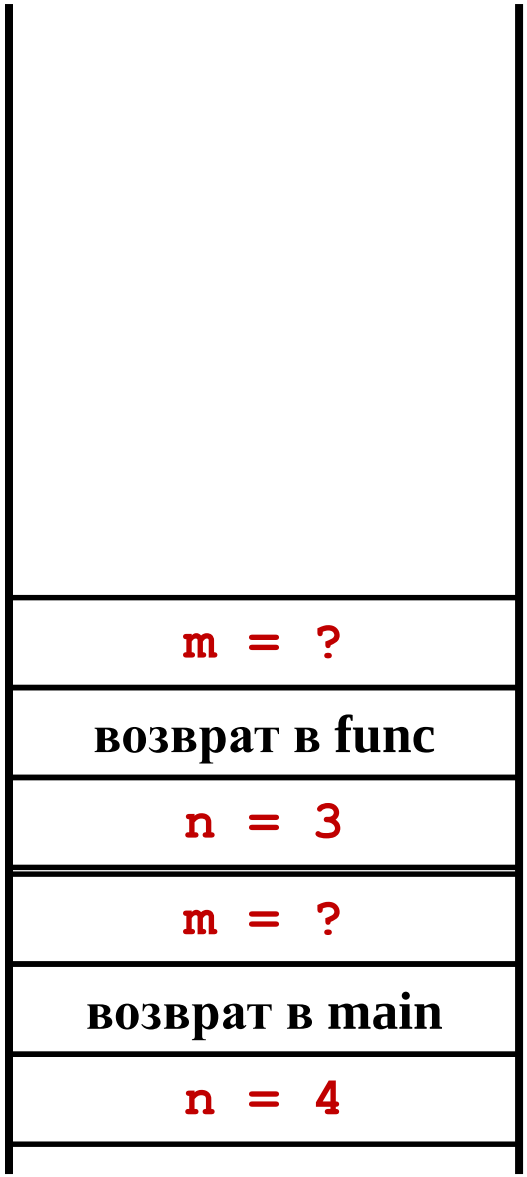
Рекурсия развертка




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

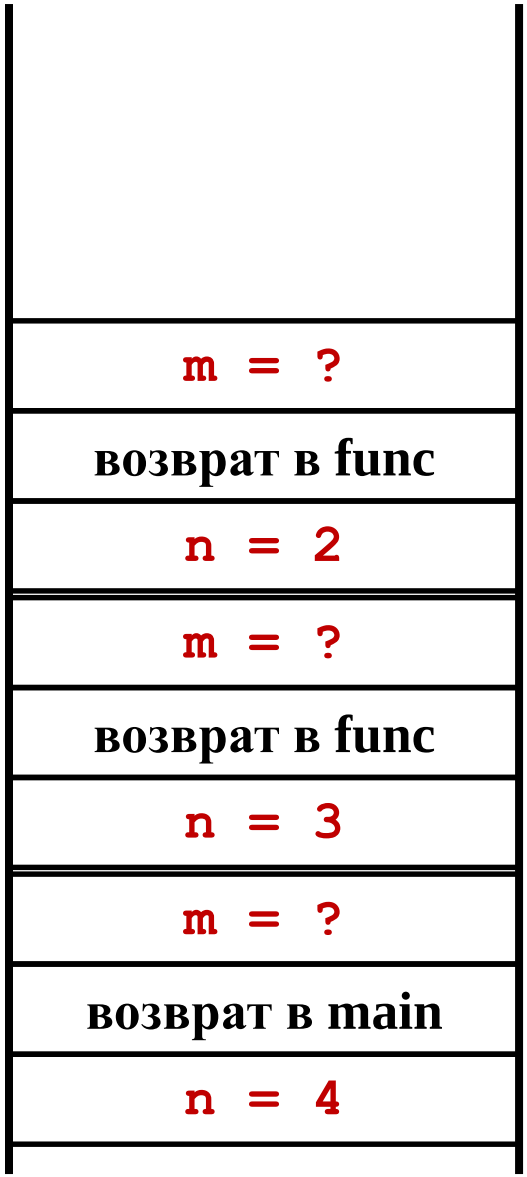
Рекурсия развертка (2)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

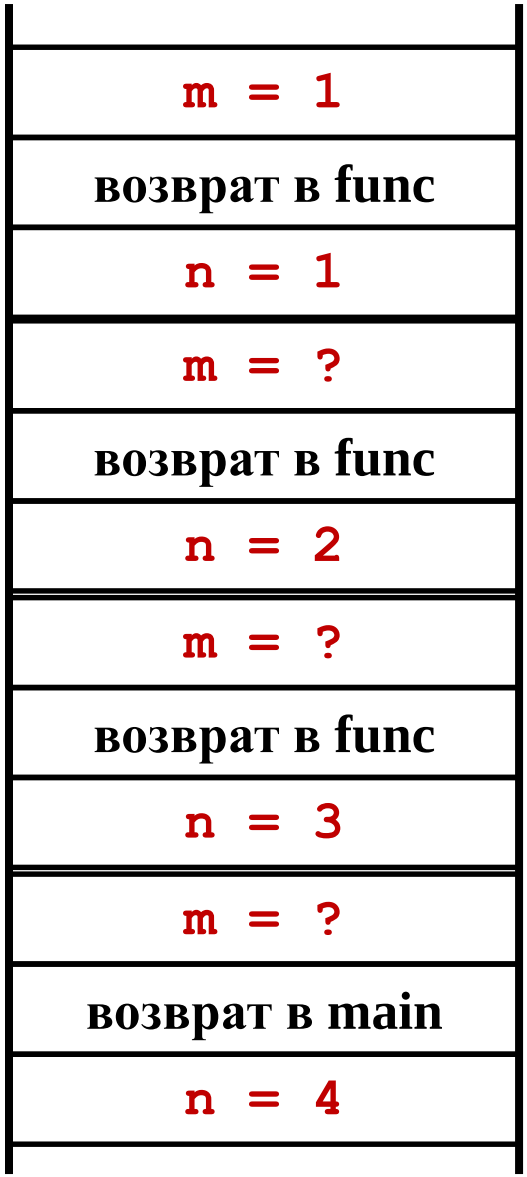
Рекурсия развертка (3)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

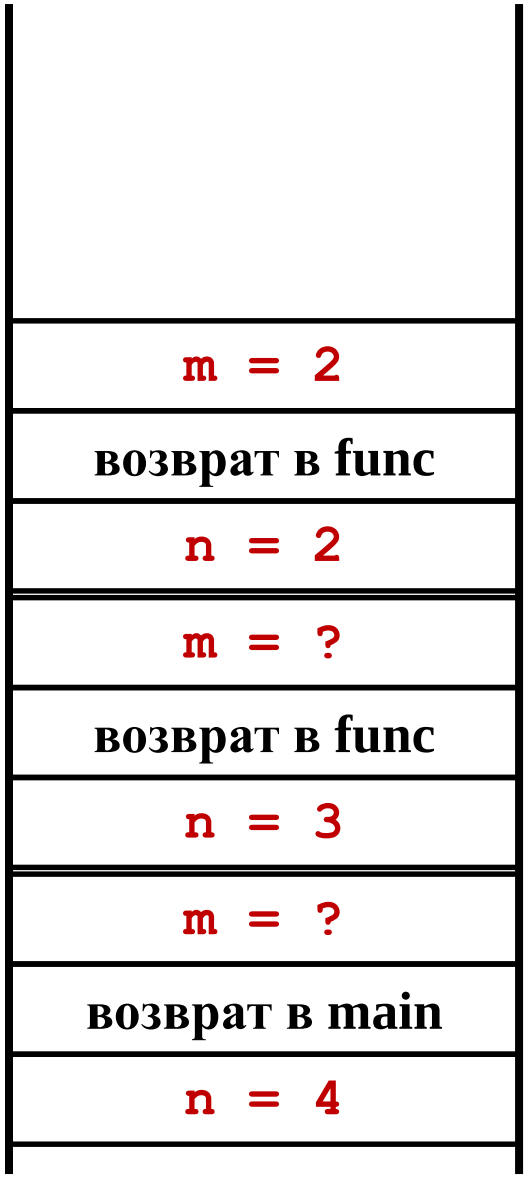
Рекурсия развертка (4)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

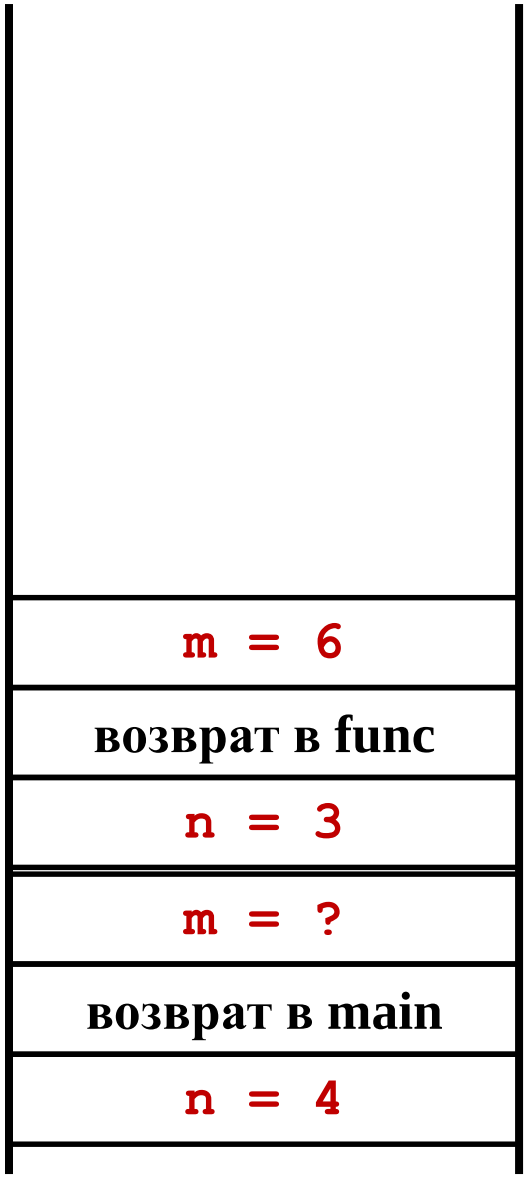
Рекурсия свертка (1)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

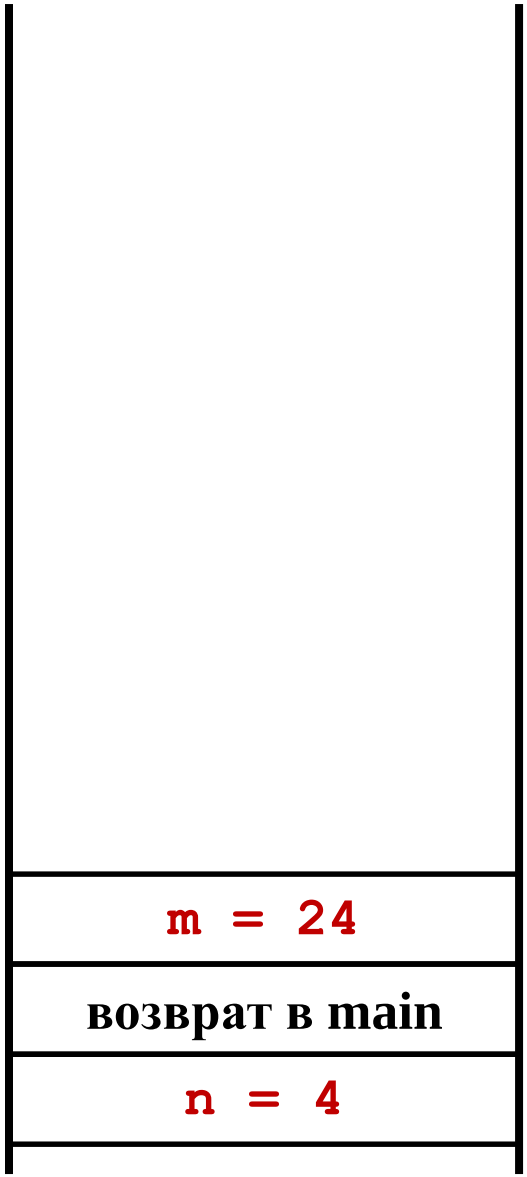
Рекурсия свертка (2)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

Рекурсия свертка (3)




```
int fact(int n){
    int m = 1;
    if( n > 1 )
        m = n*fact(n - 1);
    return m;
}

int main(){
    f = fact(4);
}
```

Отладка рекурсивной программы

```
(gdb) l fact
1           #include <stdio.h>
2
3           int fact(int n)
4           {
5               int m = 1;
6               if( n > 1 )
7                   m = n*fact(n - 1);
8               return m;
9           }
10
```

```
(gdb) b 8 if m == 1
Breakpoint 1 at 0x400518: file fact.c, line 8.
(gdb) r
Starting program: fact
Breakpoint 1, fact (n=1) at fact.c:8
8               return m;
(gdb) bt
#0  fact (n=1) at fact.c:8
#1  0x0000000000400511 in fact (n=2) at fact.c:7
#2  0x0000000000400511 in fact (n=3) at fact.c:7
#3  0x0000000000400511 in fact (n=4) at fact.c:7
#4  0x000000000040052f in main () at fact.c:12
```

Потенциальный размер стека

```
00b42000-00b63000 rw-p          [heap]
7f4d38afe000-7f4d38cbd000 r-xp /lib/libc-2.17.so
7f4d38cbd000-7f4d38ebc000 ---p /lib/libc-2.17.so
7f4d38ebc000-7f4d38ec0000 r--p /lib/libc-2.17.so
7f4d38ec0000-7f4d38ec2000 rw-p /lib/libc-2.17.so
7f4d38ec2000-7f4d38ec7000 rw-p
7f4d38ec7000-7f4d38eea000 r-xp /lib/ld-2.17.so
7f4d390c5000-7f4d390c8000 rw-p
7f4d390e7000-7f4d390e9000 rw-p
7f4d390e9000-7f4d390ea000 r--p /lib/ld-2.17.so
7f4d390ea000-7f4d390ec000 rw-p /lib/ld-2.17.so
7fff18da9000-7fff18dca000 rw-p [stack]
```

Размер: 7fff18dca000 – 7fff18da9000 = 0x21000 = 135 168_{10}

Расстояние до регионов динамических библиотек:

7fff18da9000 – 7f4d390ec000 =

0xB1DFCBD000 = 763 963 887 616 ~ 763 **ГБ**

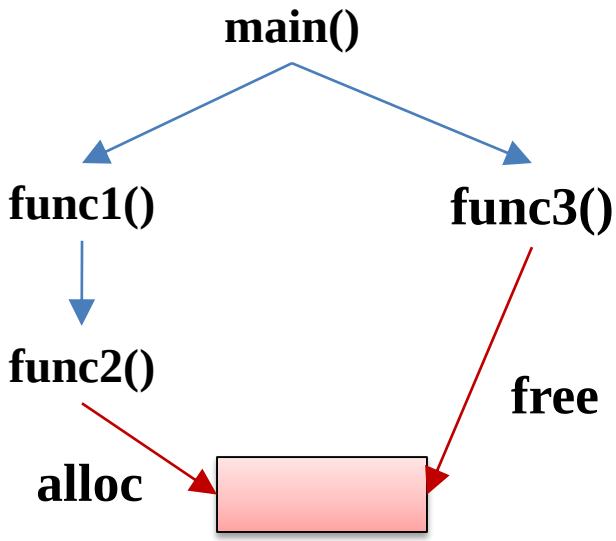
Стек и динамическая память

Структура данных стек хорошо подходит для хранения стековых кадров, т.к. до тех пор, пока текущая функция не закончит свою работу, ни одна из нижележащих функций не продолжит свое выполнение.

Динамическая память предназначена для хранения произвольных данных, время жизни которых заранее неизвестно.

Область памяти может быть выделена в одной функции и освобождена в другой.

Память продолжает существование даже тогда, когда выделившая ее функция извлекается из стека.



Динамическая память

Для реализации динамической памяти применяется структура данных "куча" (англ. heap). Куча запрашивает память у операционной системы. Эта память используется для размещения объектов, динамически созданных программой. Такие объекты продолжают свое существование до тех пор, пока не будут *явно освобождены* или программа не завершится.

В любой момент времени существования кучи вся динамическая память, разделена на *занятую* и *свободную*. Занятая память использована под размещение объектов, уже созданных и ещё не освобождённых к этому моменту времени. Из объёма свободной памяти *примитивы* работы с кучей могут выделять память под новые объекты.

Для хранения данных о принадлежности памяти к занятой или свободной обычно используется *служебная область памяти*.

Динамическая память (2)

Динамическое распределение памяти − способ выделения оперативной памяти компьютера для объектов в программе в процессе ее исполнения:

- объекты размещаются в «куче» (англ. heap);
- при создании объекта указывается размер памяти;
- в случае успеха, выделенная область памяти «изымается» из кучи и недоступна при последующих операциях выделения памяти;
- память занятая ранее под какой-либо объект может быть освобождена;
- освобождаемая память возвращается в кучу и становится доступной при дальнейших операциях выделения памяти.

Динамическая память (язык Си)

В языке Си существует четыре функции для динамического распределения памяти:

`malloc` (от англ. **m**emory **alloc**ation, выделение памяти),

`calloc` (от англ. **c**lear **alloc**ation, чистое выделение памяти)

`realloc` (от англ. **realloc**ation, перераспределение памяти).

`free` (англ. **free**, освободить)

Функции `malloc`, `calloc`, `realloc` обеспечивают выделение памяти, функция `free` – освобождение памяти, возвращенной любой из функций ее выделения.

```
#include <stdlib.h>
void *malloc (size_t size);
void *calloc (size_t num, size_t size);
void *realloc(void *block, size_t size);
void *free(void *block);
```

Динамическая память (3)

Для размещения и удаления динамических объектов используются примитивы «создать объект» (например, `malloc`) и «удалить объект» (например, `free`).

Кроме того, перед началом работы программы выполняется *инициализация кучи*, в ходе которой вся изначально выделенная под кучу память отмечается как свободная.

Служебная информация

Смещение	0	0	0	0	0	0	0	0	0
Размер	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
+ 0										
+10										
+20										
+30										

Размещение в куче (1)

```
char *p1 = malloc(4);
```

Служебная информация

Смещение	**0**	0	0	0	0	0	0	0	0
Размер	**4**	0	0	0	0	0	0	0	0

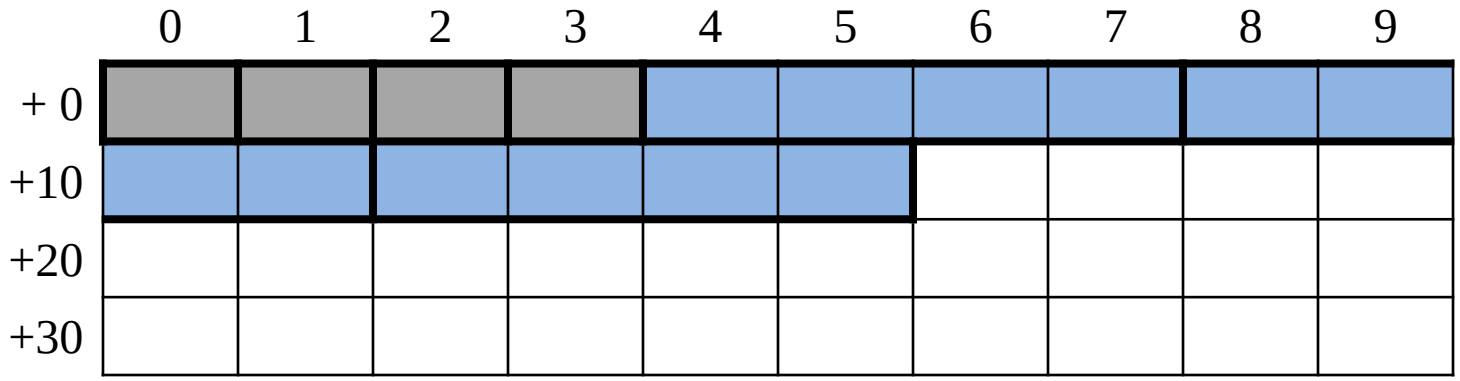
	0	1	2	3	4	5	6	7	8	9
+ 0										
+10										
+20										
+30										

Размещение в куче (2)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
```

Служебная информация

Смещение	**0**	**4**	0	0	0	0	0	0	0
Размер	**4**	**12**	0	0	0	0	0	0	0

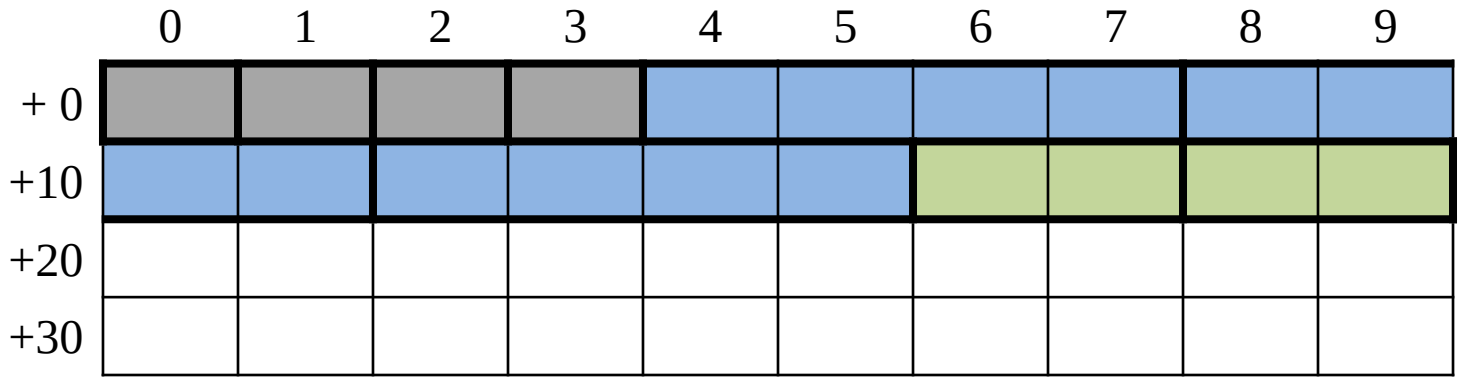


Размещение в куче (3)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
```

Служебная информация

Смещение	**0**	**4**	**16**	0	0	0	0	0	0
Размер	**4**	**12**	**4**	0	0	0	0	0	0

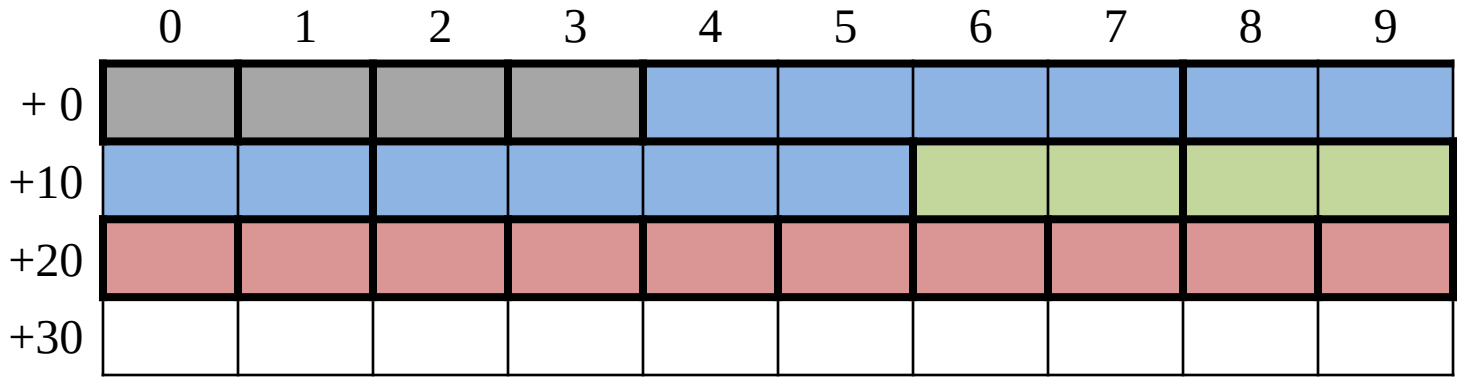


Размещение в куче (4)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
char *p4 = malloc(10*sizeof(char));
```

Служебная информация

Смещение	**0**	**4**	**16**	**20**	0	0	0	0	0
Размер	**4**	**12**	**4**	**10**	0	0	0	0	0

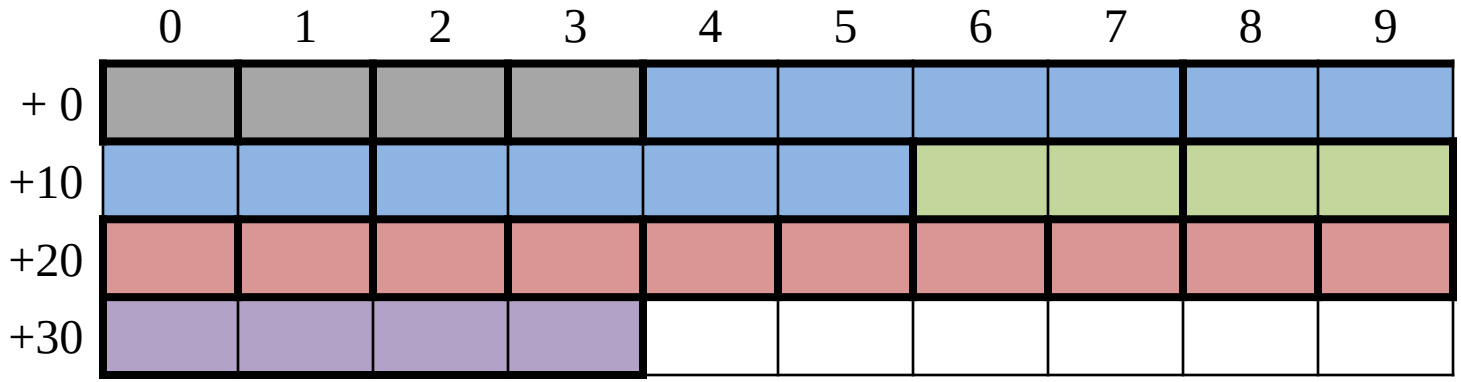


Размещение в куче (5)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
char *p4 = malloc(10*sizeof(char));
int *p5 = malloc(sizeof(int));
```

Служебная информация

Смещение	**0**	**4**	**16**	**20**	**30**	0	0	0	0
Размер	**4**	**12**	**4**	**10**	**4**	0	0	0	0

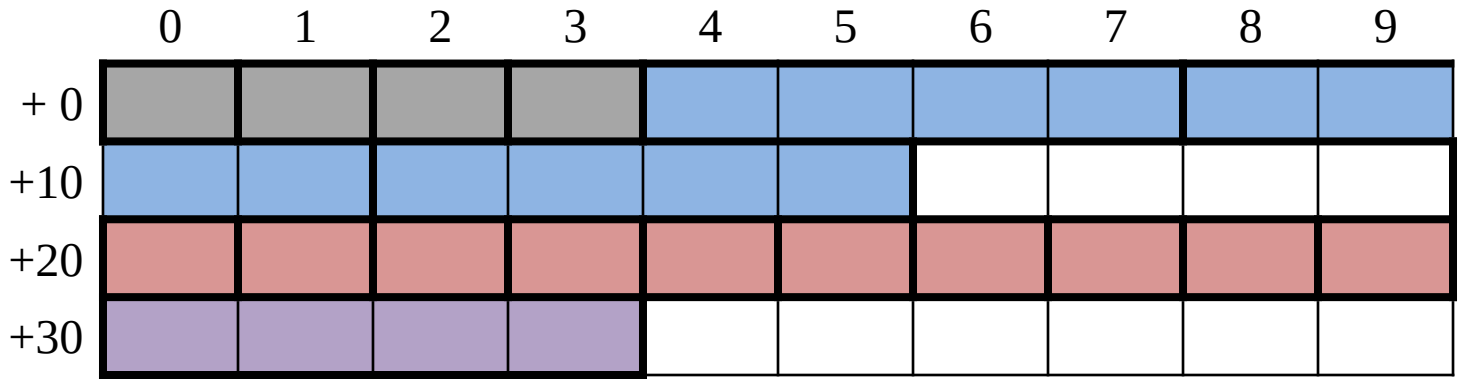


Освобождение размещения (1)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
char *p4 = malloc(10*sizeof(char));
int *p5 = malloc(sizeof(int));
free(p3);
```

Служебная информация

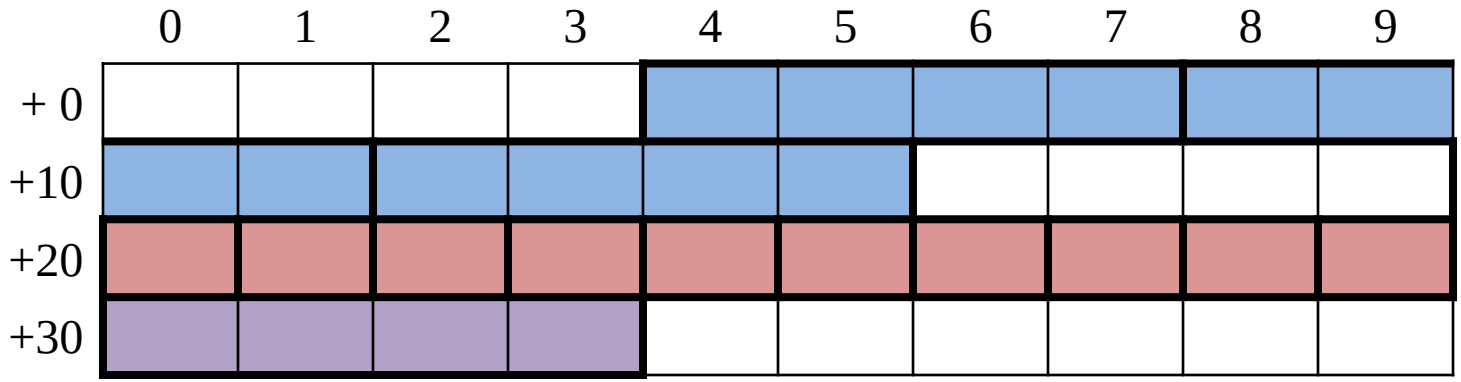
Смещение	**0**	**4**	**20**	**30**	0	0	0	0	0
Размер	**4**	**12**	**10**	**4**	0	0	0	0	0



Освобождение размещения (2)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
char *p4 = malloc(10*sizeof(char));
int *p5 = malloc(sizeof(int));
free(p3);
free(p1);
```

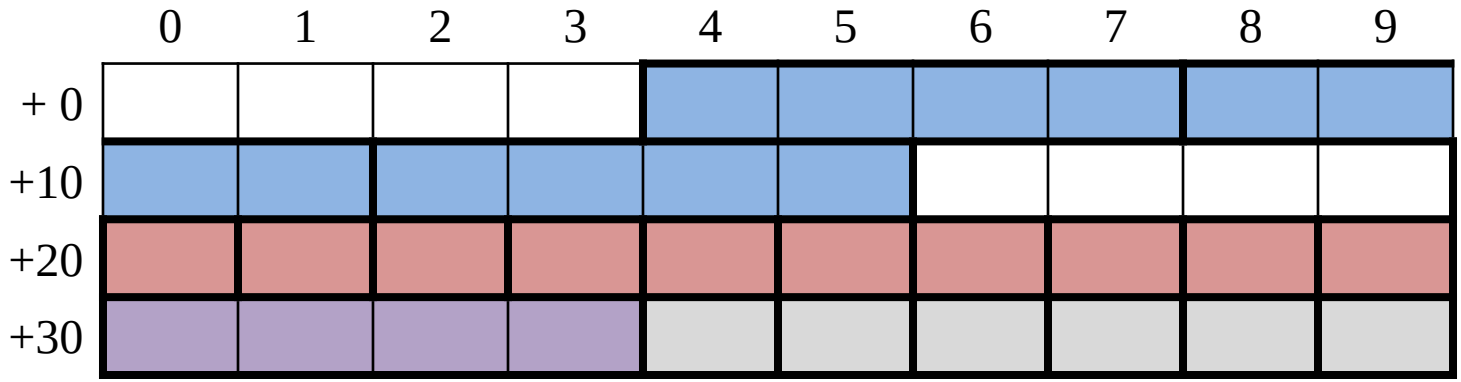
Смещение	**4**	**20**	**30**	0	0	0	0	0	0
Размер	**12**	**10**	**4**	0	0	0	0	0	0



Освобождение размещения (2)

```
char *p1 = malloc(4);
int *p2 = malloc(3*sizeof(int));
short *p3 = malloc(2*sizeof(short));
char *p4 = malloc(10*sizeof(char));
int *p5 = malloc(sizeof(int));
free(p3);
free(p1);
p1 = malloc(6);
```

Смещение	**4**	**20**	**30**	**34**	0	0	0	0	0
Размер	**12**	**10**	**4**	**6**	0	0	0	0	0



realloc: перераспределение памяти

Как было сказано ранее, в языке Си существует примитив, позволяющий *изменить размер выделенной области памяти* с сохранением имеющихся значений(!): realloc:

```
int *p2 = malloc(3*sizeof(int));
...
p2 = realloc(p2, 4*sizeof(int));
```

Смещение	**0**	**4**	**16**	0	0	0	0	0	0
Размер	**4**	**12**	**4**	0	0	0	0	0	0

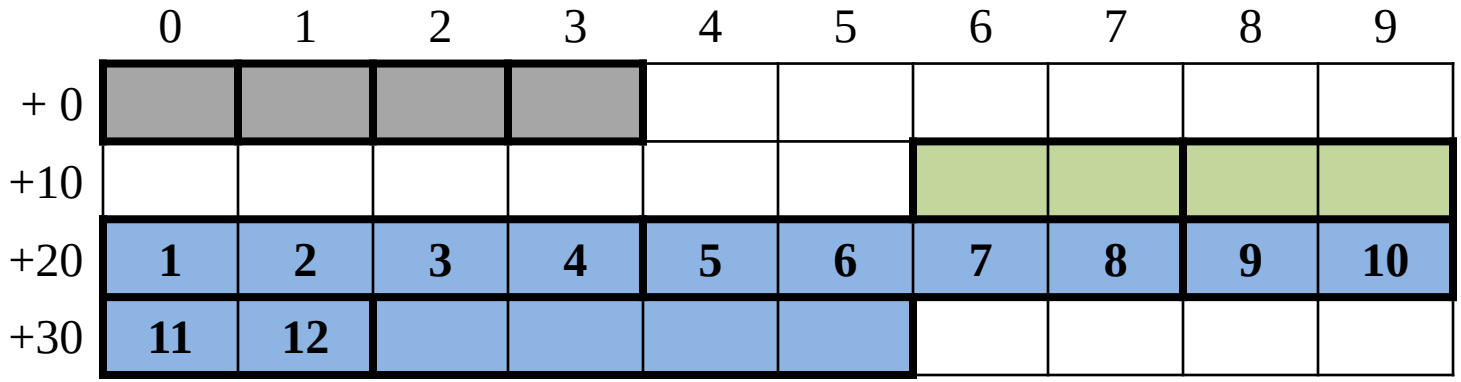
	0	1	2	3	4	5	6	7	8	9
+ 0					**1**	**2**	**3**	**4**	**5**	**6**
+10	**7**	**8**	**9**	**10**	**11**	**12**				
+20										
+30										

realloc: перераспределение памяти (2)

Как было сказано ранее, в языке Си существует примитив, позволяющий *изменить размер выделенной области памяти* с сохранением имеющихся значений(!): realloc:

```
int *p2 = malloc(3*sizeof(int));
...
p2 = realloc(p2, 4*sizeof(int));
```

Смещение	**0**	**20**	**16**	0	0	0	0	0	0
Размер	**4**	**16**	**4**	0	0	0	0	0	0



realloc: перераспределение памяти (3)

Как было сказано ранее, в языке Си существует примитив, позволяющий *изменить размер выделенной области памяти* с сохранением имеющихся значений(!): realloc:

```
char *p1 = malloc(4);
...
p1 = realloc(p1, 8);
```

Смещение	**0**	**20**	**16**	0	0	0	0	0	0
Размер	**4**	**16**	**4**	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
+ 0	**a**	**b**	**c**	**d**						
+10										
+20	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
+30	**11**	**12**								

realloc: перераспределение памяти (3)

Как было сказано ранее, в языке Си существует примитив, позволяющий *изменить размер выделенной области памяти* с сохранением имеющихся значений(!): realloc:

```
char *p1 = malloc(4);
...
p1 = realloc(p1, 8);
```

Смещение	**0**	**20**	**16**	0	0	0	0	0	0
Размер	**8**	**16**	**4**	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
+ 0	**a**	**b**	**c**	**d**						
+10										
+20	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
+30	**11**	**12**								

Динамически-расширяемый массив

Динамическое выделение памяти характеризуется **существенными накладными расходами**, которые связаны с **поиском** свободного блока памяти, подходящего под запрос, т.е. размер которого близок к указанному в функции malloc.

Накладные расходы при перераспределении памяти могут быть существенно выше, так как помимо поиска подходящей области памяти необходимо также выполнить **копирование** содержимого из старой области в новую.

Для того, чтобы снизить указанные накладные расходы при организации динамически расширяемых массивов часто используют следующую стратегию:

1. Изначальный размер массива – x.

2. Если размера массива не достаточно для хранения денных – увеличить его размер вдвое: $x = 2 \cdot x$.

Таким образом, количество перераспределений логарифмически (а не линейно!) зависит от размера массива.

Потенциальный размер динамической памяти

```
00b42000-00b63000 rw-p          [heap]
7f4d38afe000-7f4d38cbd000 r-xp /lib/libc-2.17.so
7f4d38cbd000-7f4d38ebc000 ---p /lib/libc-2.17.so
7f4d38ebc000-7f4d38ec0000 r--p /lib/libc-2.17.so
7f4d38ec0000-7f4d38ec2000 rw-p /lib/libc-2.17.so
7f4d38ec2000-7f4d38ec7000 rw-p
7f4d38ec7000-7f4d38eea000 r-xp /lib/ld-2.17.so
7f4d390c5000-7f4d390c8000 rw-p
7f4d390e7000-7f4d390e9000 rw-p
7f4d390e9000-7f4d390ea000 r--p /lib/ld-2.17.so
7f4d390ea000-7f4d390ec000 rw-p /lib/ld-2.17.so
7fff18da9000-7fff18dca000 rw-p [stack]
```

Размер: 0x00b63000– 0x00b42000 = 0x21000 = 135 168_{10}

Расстояние до регионов динамических библиотек:

7f4d38afe000 – 0x00b63000=

0x7F4D37F9B000 = 139 969 628 319 744 ~ 139 **ТБ**

Классы памяти переменных

Время жизни и область видимости переменной в языке Си определяется **классом памяти** переменной. Существуют следующие классы:

`auto` (**сегмент стека**) - локальные переменные, память для которых выделяется при входе в составной оператор, и освобождается при выходе из него.

`register` (**регистры процессора, если доступны**) – аналогичен классу **`auto`**, но, если это возможно, переменная будет размещена в процессорном регистре.

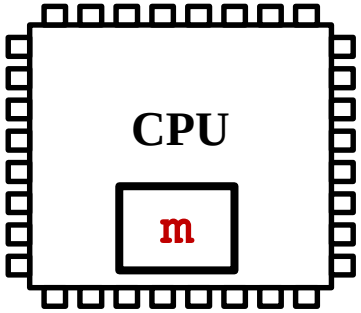
`static` (**сегмент данных**) – переменные (локальные или внешние), существующие в течение всего выполнения программы.

`extern` (**сегмент данных**) – глобальные переменные, используются для связи между функциями, в том числе независимо скомпилированными, которые могут находиться в различных файлах.

Классы памяти **auto** и **register**

Расположение	Стек вызовов / регистры процессора
Время жизни	Однократное выполнение блока операторов
Область видимости	В рамках блока операторов

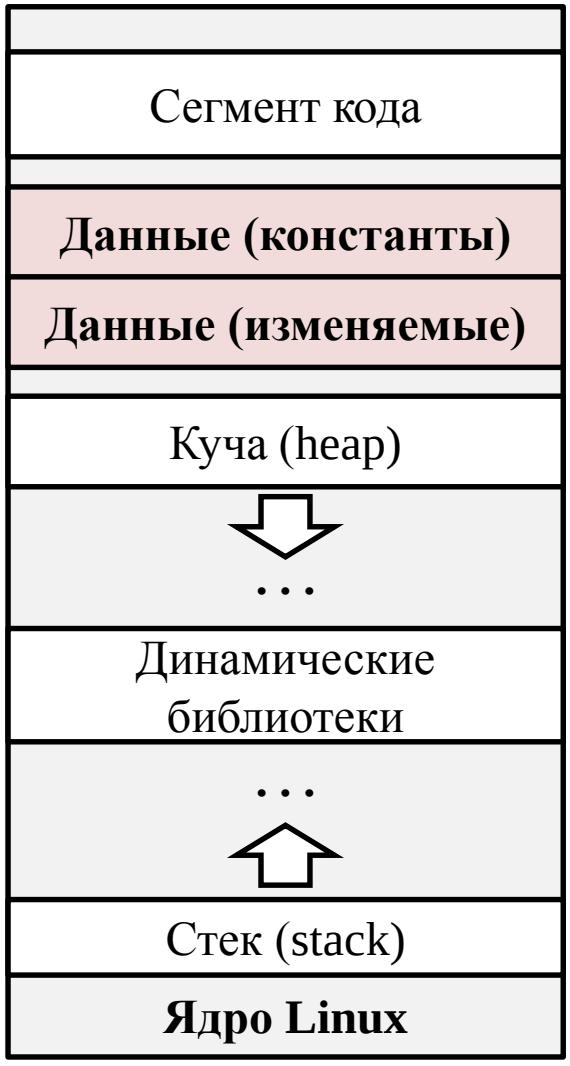
```
int  f(int i)
{
  int r = i;
  auto int k = 0;
  register int m;
  ...
}
```



Класс памяти **static** (локальные переменные)

Расположение	Сегмент данных
Время жизни	Все время исполнения программы
Область видимости	В рамках блока операторов

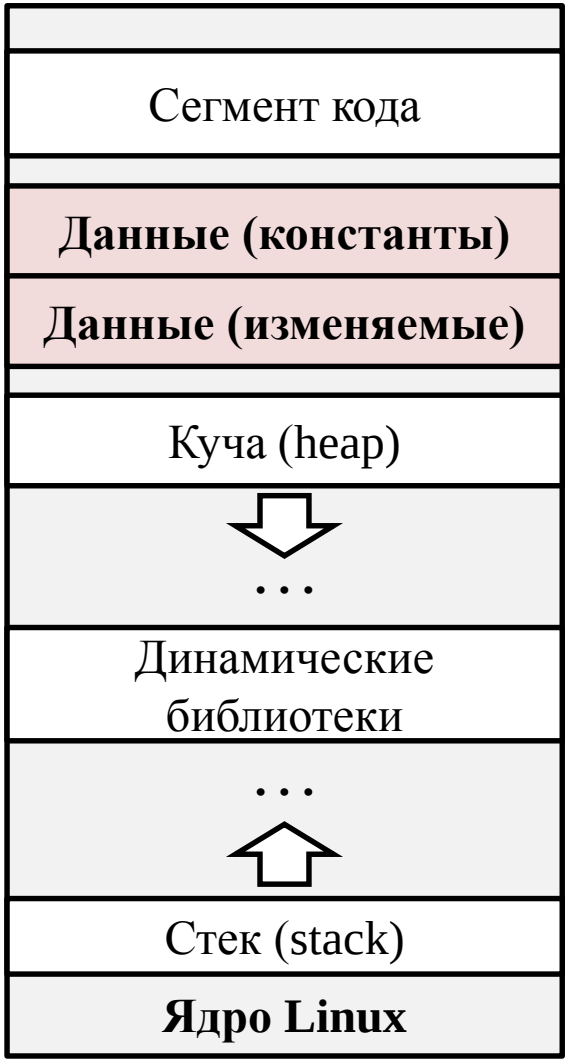
```
int  f(int i)
{
  static int r;
  static int m = 0;
...
}
```



Класс памяти **static** (внешние переменные)

Расположение	**Сегмент данных**
Время жизни	Все время исполнения программы
Область видимости	В рамках файла, начиная с объявления

```
static int r;
int  f(int i)
{
   ...
   r = i;
   ...
}
```



Глобальные переменные (внешние без класса памяти)

Расположение	Сегмент данных
Время жизни	Все время исполнения программы
Область видимости	1. В рамках текущего файла, начиная с места объявления 2. Область видимости может быть распространена на другие файлы с использованием класса памяти `extern`.

```
int glob = 18;

void f(){
  ...
}
```

Класс памяти **extern** (внешние переменные)

Расположение	Сегмент данных
Время жизни	Все время исполнения программы
Область видимости	В рамках файла, начиная с объявления

```
int glob = 18;

void f(){
  printf("GV=%d",glob);
  glob++;
  return;
}
```

functions.c

```
  ...
void f();
extern int glob;
int main(){
  printf("1:%d",glob);
  f();
  printf("2:%d",glob);
}
```

main.c

Класс памяти **extern** (локальные переменные)

Расположение	Сегмент данных
Время жизни	Все время исполнения программы
Область видимости	В рамках блока операторов

```
int glob = 18;

void f(){
   ...
}
```

func1.c

```
   ...
int f2(){
   extern int glob;
   ...
}
```

func2.c

Разрешение конфликтов имен переменных

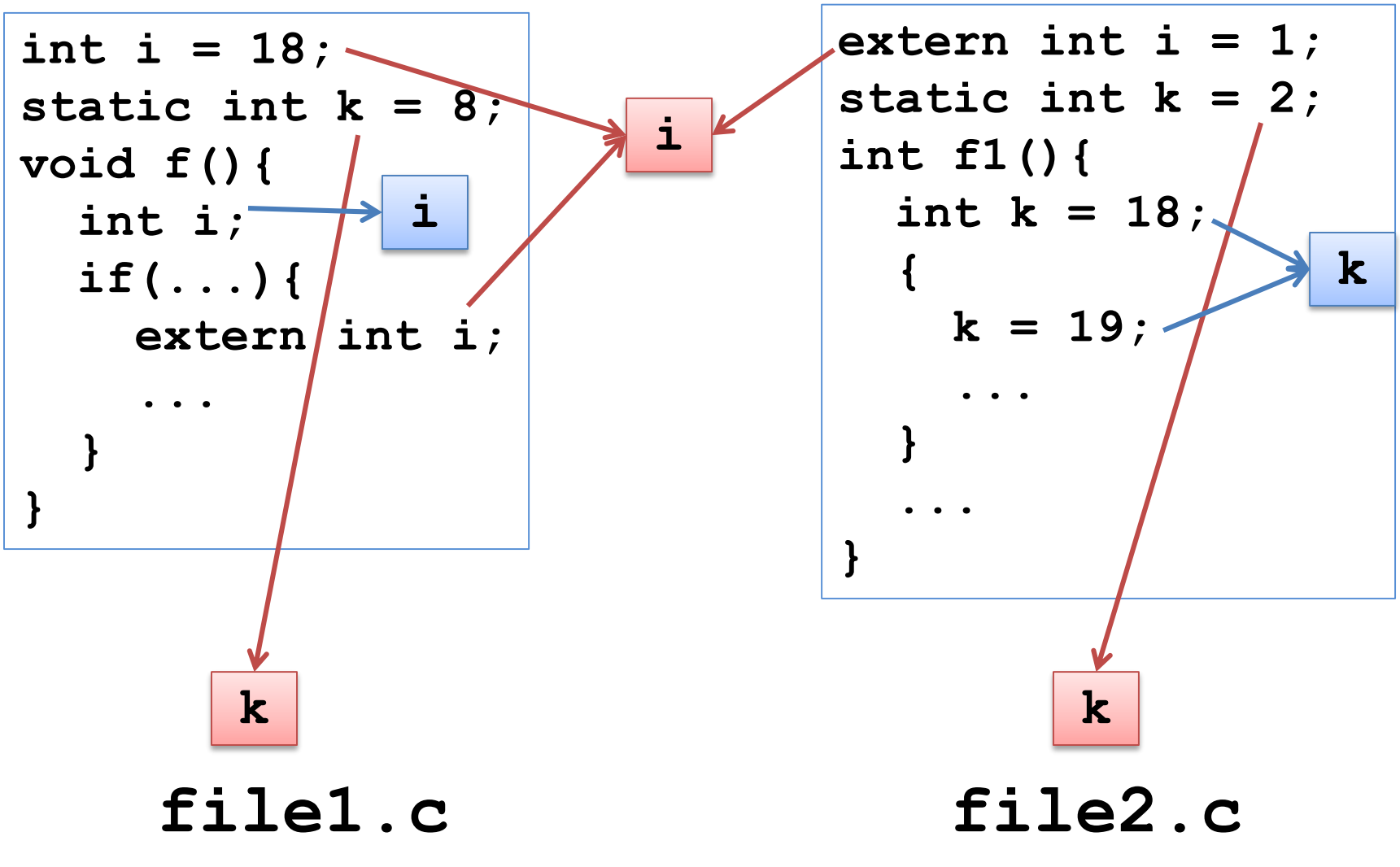
При разрешении конфликтов имен компилятор руководствуется следующими правилами:

1. Переменные с идентичной областью видимости **не могут** иметь одинаковых имен.

2. Если области видимости переменных **пересекаются**, переменная с меньшей областью видимости является более приоритетной.

3. Если области видимости не пересекаются, конфликта нет.

Пример разрешения конфликтов имен переменных



Классы памяти функций

В связи с тем, что время жизни функции всегда равно времени выполнения программы, количество классов для функций меньше, чем для переменных:

`extern` – класс памяти по умолчанию, указывать не обязательно;

`static` – класс памяти, предусматривающий ограничение области видимости функции в рамках одного файла.

Функции класса памяти **extern**

```
int func1(int x);

void func2(){
  int i = 10;
  if(...){
    func1(i);
    ...
  }
}
```

f1.c

```
int func1(int x){
    . . .
}
```

f2.c

```
int func1(int x);
int func3(){
   func1(5);
}
```

f3.c

```
gcc -Wall -o prog f1.c f2.c f3.c
```

Функции класса памяти **static**

```
static
int func1(int x){
    ...
}
void func2(){
   int i = 10;
   if(...){
      func1(i);
      ...
   }
}
```

f1.c

```
int func1(int x){
    . . .
}
```

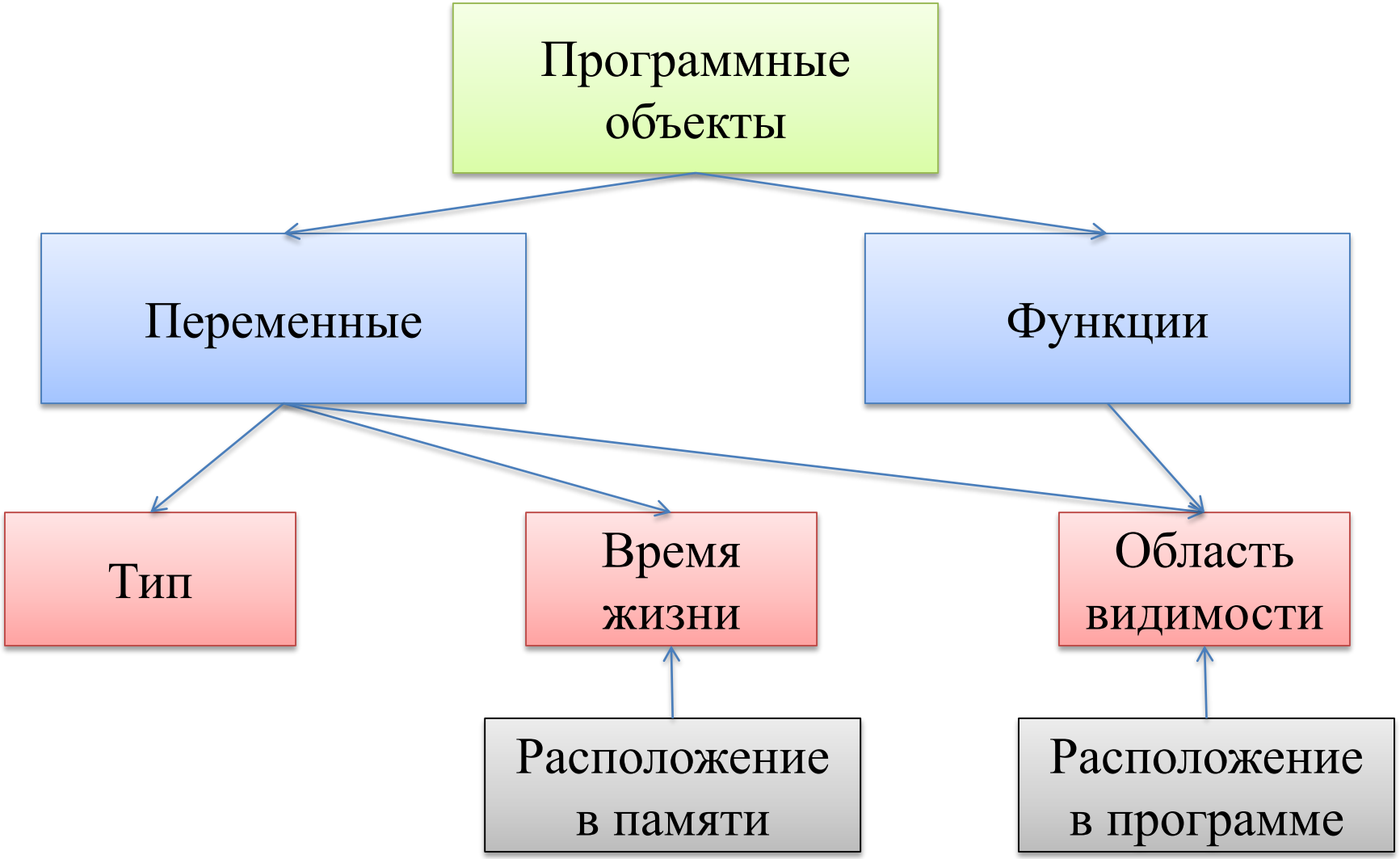
f2.c

```
int func1(int x);
int func3(){
    func1(5);
}
```

f3.c

```
gcc -Wall -o prog f1.c f2.c f3.c
```

Характеристики программных объектов



Многофайловые программы

- Для структурирования исходного кода программы ее текст может быть распределен по нескольким файлам.
- Минимальные неделимые единицы разбиения программы: **функции** и **внешние переменные**.
- Только один файл может содержать функцию main
- Программные объекты могут быть **глобальными в рамках файла**. В этом случае они не доступны из функций, расположенных в других файлах. Локальными являются функции и внешние переменные, имеющие класс памяти static.
- Программные объекты могут быть **глобальными в рамках программы**, т.е. доступными из любой ее точки. Глобальными являются функции и внешние переменные с классом памяти extern.

Пример программы, состоящей из нескольких файлов с исходным кодом

```
int sumsub(int *i,int j)
{
  int k = *i
  *i = k - j;
  return k + j;
}
```

sumsub.c

```
int sumsub(int*,int);
int main(){
   int s, m=3;
   s = sumsub(&m, 5);
   printf("%d",m);
}
```

main.c

```
gcc –Wall –o sum sumsub.c main.c
```

Программа подлежащая разбиению на файлы

```
prog.c:
#include <stdio.h>
int input(...){
    // Some input actions
    ...
}
int update(...){
    // Update data
    ...
}
int process(...){
    // Input processing
    ...
}
```

```
int main()
{
    ...
    input(...);
    ...
    update(...);
    ...
    process(...);
}
```

Распределение программы по файлам

io.h:

```
#ifndef IO_H
#define IO_H
#include
"process.h"

int input(...);

#endif
```

io.c:

```
#include
<stdio.h>
#include "io.h"

int input(...){
    //        Input
actions
    ...
}
```

main.c:

```
#include "io.h"
#include "process.h"

int main()
{
    ...
    input(...);
    ...
    update(...);
    ...
    process(...);
}
```

process.h:

```
#ifndef PROCESS_H
#define PROCESS_H

int update(...);
int process(...);

#endif
```

process.c:

```
#include <stdio.h>
#include
"process.h"

int update(...){
// Update input
    ...
}
int process(...){
// Final processing
    ...
}
```

Формат заголовочного файла

processing.h:

```
1  #ifndef  <VAR_NAME>  //  Уникальное  имя  препроцессорной
переменной
2 #define <VAR_NAME> // То же имя переменной
3
4 #include <stdio.h> // Подключение необходимых файлов
5 // Список прототипов
6 int function1(int i, char j, float t);
7 float function2(int mas[], char t);
8
9 #endif // окончание ifdef в строке 1
```

Строки 1,2 и 9 служат для защиты от
многократного включения заголовочного файла.

Разработка многофайловых программ

io.h:

```
#ifndef IO_H
#define IO_H
#include
“process.h”
int input(...);
#endif
```

process.h:

```
#ifndef PROCESS_H
#define PROCESS_H
int update(...);
int process(...);
#endif
```

main.c:

```
#include
<stdio.h>
#include “io.h”
#include
“process.h”
int main()
{
   ...
}
```

main.i:

```
00 #include <stdio.h>
01 #ifndef IO_H
02 #define IO_H
03     #ifndef PROCESS_H
04     #define PROCESS_H
05     int update(...);
06     int process(...);
07     #endif
08 int input(...);
09 #endif
10 #ifndef PROCESS_H // ложь, опр. в с.03
11 #define PROCESS_H
12 int update(...);
13 int process(...);
1$ #endif
15
16 int main()
17 {
18    ...
19 }
```