



© Кафедра вычислительных систем ФГБОУ ВПО «СибГУТИ»

"ПРОГРАММИРОВАНИЕ"

Типизация данных в языках высокого уровня

Преподаватель:

Перышкова Евгения Николаевна



Память в вычислительной технике

- Компьютерная память представляет собой огромный массив(набор) байтов (8-мибитных ячеек).
- Для хранения полезной информации в памяти компьютера требуется "разметить" ее, т.е. создать виртуальные границы между байтами, отделяющие одни данные от других.

Этот процесс похож на организацию офисного помещения с помощью перегородок.

Одно и то же помещение может
быть оборудовано под различное
использование





Простейшие типы данных

Вычислительная техника (ВТ) предоставляет программисту возможность хранения данных, имеющих числовую природу: целые или вещественные числа.

Языки программирования высокого уровня предоставляют программисту набор базовых типов данных, которые определяют:

1. Формат хранения информации, поддерживаемый ВТ:

- беззнаковые целые числа (двоичная запись числа);
- знаковые целые числа (дополнительный код);
- вещественные числа (формат с плавающей точкой).

2. Размер ячейки памяти:

- 1 (char), 2 (short), 4 (int), 8 (long int) байт для знаковых и беззнаковых целых;
- 4 (float), 8 (double), 12 (long double) байт для вещественных.

Простейшие типы данных (2)

Неразмеченная область памяти:

[illegible]

```
#include <stdio.h>
int main()
{
    char c;
    unsigned short s;
    int i, j;
    float f;
    double d;
}
```

В данной программе определен набор переменных, который задает разметку доступной памяти следующим образом:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
c	s		i				j				f						d						



Символы [ASCII коды]

Для хранения символов в вычислительной технике используются однобайтовые ячейки, содержащие коды символов. Кодирование позволяет сохранить символьные данные в виде чисел.

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.		!	"	#	\$	%	&	'	()	*	+	,	—	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL



Сложные типы данных [массивы]

- Языки высокого уровня представляют программисту возможность также объединять наборы однотипных данных в массивы.
- Массивы предоставляют программисту инструменты, необходимые для их автоматизированной обработки:

```
int a0, a1, a2, a3, a4;  
a0 = 1;  
a1 = 2;  
a2 = 3;  
a3 = 4;  
a4 = 5;
```

```
int a[5], i;  
for(i=0; i<5; i++){  
    a[i] = i + 1;  
}
```

- Массивы представляют собой набор ячеек, расположенных в памяти непрерывно друг за другом. Это обеспечивает простоту доступа к отдельным элементам.

	1	2	3	4	5			
a[0]	a[1]	a[2]	a[3]	a[4]				



Сложные типы данных [строки]

- Для хранения текстовой информации в языке Си используются строки.
- Строка представляет собой массив символов. Для обозначения конца строки используется специальный символ с ASCII кодом 0, который называется нуль-терминатором.

```
char str[16] = "Hello world!";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Н	е	л	л	о		w	o	r	l	d	!	\0				...

- Нуль-терминатор позволяет определять количество ячеек, использованных под полезную информацию. Например, на приведенном выше рисунке полезно используются только первые 12 из 16 доступных байт.
- Для работы со строками в языке Си предусмотрен широкий набор функций, каждая из которых требует наличия нуль-терминатора в строке.



- ## Неразмеченная область памяти:

```
struct person {
    char name[8];
    char last_name[10];
    unsigned short age;
} x = { "Jack", "Brown", 25 };
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
J	a	c	k	\0				B	r	o	w	n	\0					25					
name[8]								last_name[10]										age					





Синтаксис описания структуры в языке Си

Структура =

```
"struct" [Тэг] "{"  
    {Тип " " Идентиф{ " ," Идентиф } ";" }  
    "}" [ Объект { " ," Объект } ] ";"
```

Объект = Идентиф [**"=" Значение**].

Тип = БазовыйТип | Структура | Объединение
| Массив | Указатель.

```
struct {  
    char name[64];  
    char lastname[64];  
    char patronymic[64];  
    unsigned char age;  
    struct address addr;  
} var;
```

```
struct address{  
    char counrty[64];  
    int index;  
    char city[64];  
    char street[64];  
    int build, flat;  
};
```



Синтаксис описания структуры в языке Си (2)

Структура =

```
"struct" [Тэг] "{"  
    {Тип " " Идентиф{ " ," Идентиф } ";" }  
    "}" [ Объект { " ," Объект } ] ";"
```

Объект = Идентиф [**"=" Значение**].

Тип = БазовыйТип | Структура | Объединение
| Массив | Указатель.



```
struct address{  
    char country[4];  
    char series[3];  
    unsigned short num;  
    unsigned short region;  
};
```



Инициализация структур

1. Инициализация в порядке следования элементов

```
struct address{  
    char counrty[64];  
    int index;  
    char city[64];  
    char street[64];  
    int build, flat;  
} a = { "Russia", 630102, "Novosibirsk", "Kirova", 86, 402 };
```

2. Инициализация элементов в произвольном порядке

```
struct address{  
    char counrty[64];  
    int index;  
    char city[64];  
    char street[64];  
    int build, flat;  
} b = { .index = 630102, .street = "Kirova", .build = 86};
```



Допустимые операции

```
#define N 100
struct array{
    int max, cur;
    int mas[N];
} a1, a2 = {N,4,{1,2,3,4}};
```

- присваивание переменных-структур: `a1 = a2;`
- взятие адреса: `struct array *p = &a1;`
- обращение к элементам структуры (операции "." и "->"):

```
a1.cur = 10; a1.cur++; // переменная
p->cur = 10; p->cur++; // указатель
```

- Применение операции sizeof:

```
int s = sizeof(a1), s1 = sizeof(*p);
```



Практический пример:

Массив с проверкой выхода за границы

```
struct safearr{  
    unsigned int len;  
    int array[1024];  
};
```

```
void init(struct safearr *arr){  
    arr->len = 0;  
}
```

```
int add_elem(struct safearr *arr, int val){  
    int max = sizeof(arr->array)/sizeof(arr->array[0]);  
    if( arr->len + 1 < max ){  
        arr->array[arr->len++] = val;  
        return 0;  
    }  
    return -1;  
}  
  
int get_elem(struct safearr *arr, int index, int *val){  
    if( arr->len > index ){  
        *val = arr->array[index];  
        return 0;  
    }  
    return -1;  
}
```

Язык Си не предусматривает проверку границ массива. Однако, этот функционал можно реализовать самостоятельно



Память в вычислительной технике (повтор)

- Компьютерная память представляет собой огромный массив(набор) байтов (8-мибитных ячеек).
- Для хранения полезной информации в памяти компьютера требуется "разметить" ее, т.е. создать виртуальные границы между байтами, отделяющие одни данные от других.

Этот процесс похож на организацию офисного помещения с помощью перегородок.

Одно и то же помещение может быть оборудовано под различное использование





Адресация данных

- Для формирования виртуальных границ между областями компьютерной памяти, используемой для хранения различных данных, **используются адреса**.
- **Адрес** – порядковый номер первого байта области памяти.

Рассмотрим в качестве примера следующее расположение ячеек данных в памяти.

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
c	s		i				f							d					

В данном случае переменные имеют следующие адреса:

&c == 10, &s == 11, &i == 13, &f == 17, &d == 21

Адрес задает только начало области, в которой расположены данные. Для того, чтобы определить последний байт области используется информация о размере типа данного.



Адресация данных (2)

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
c	s		i				f							d					

В данном случае переменные имеют следующие адреса:

$\&c == 10$, $\&s == 11$, $\&i == 13$, $\&f == 17$, $\&d == 21$

Адрес задает только начало области, в которой расположены данные. Для того, чтобы определить последний байт области используется информация о размере типа данного:

c: $[\&c, \&c + (\text{sizeof}(\text{char}) - 1)] = [10, 10]$

s: $[\&s, \&s + (\text{sizeof}(\text{short}) - 1)] = [11, 12]$

i: $[\&i, \&i + (\text{sizeof}(\text{int}) - 1)] = [13, 16]$

f: $[\&f, \&f + (\text{sizeof}(\text{float}) - 1)] = [17, 20]$

d: $[\&d, \&d + (\text{sizeof}(\text{double}) - 1)] = [21, 28]$



Выравнивание данных

Центральные процессоры в качестве основной единицы при работе с памятью используют машинное слово, размер которого может быть различным. Как правило, машинное слово равно 4 или 8 байт.

<http://www.ibm.com/developerworks/power/library/pa-dalign/>

Некоторые модели процессоров не могут обращаться к данным в памяти, нарушающим границы машинных слов. У других такое обращение занимает больше времени. Рассмотрим предыдущий пример, считая, что машинное слово составляет 4 байта.

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

c	s	i	f	d
---	---	---	---	---

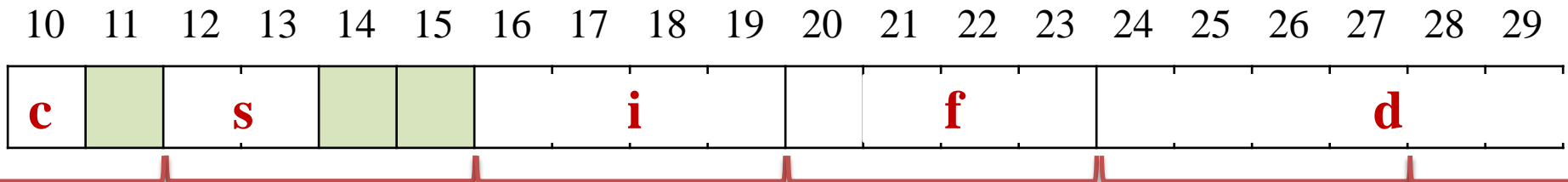
Для считывания содержимого ячеек **s**, **i** и **f** потребуется ДВА обращения к 4-хбайтной шине данных, в то время как при более эффективном расположении достаточно одного.

Для чтения ячейки **d** потребуется 3 обращения к шине данных вместо двух.



Выравнивание данных (2)

Более эффективное (с точки зрения скорости доступа) будет выглядеть следующим образом:



Такое расположение достигается за счет следующего правила:

Адрес ячейки должен быть кратен ее размеру!

Недостатком такого подхода является появление неиспользуемых **выравнивающих байтов** (padding bytes), выделенных на рисунке. Однако соотношение скорость/объем у данного решения выше, чем у рассмотренного ранее.

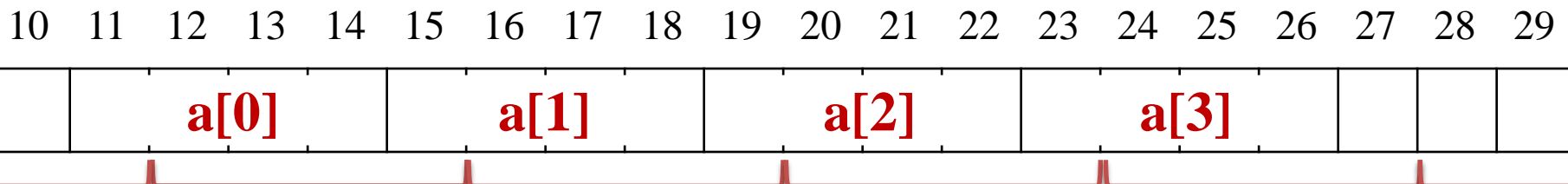
Также можно добиться более оптимального размещения, если переменные будут упорядочены по **убыванию размера**.



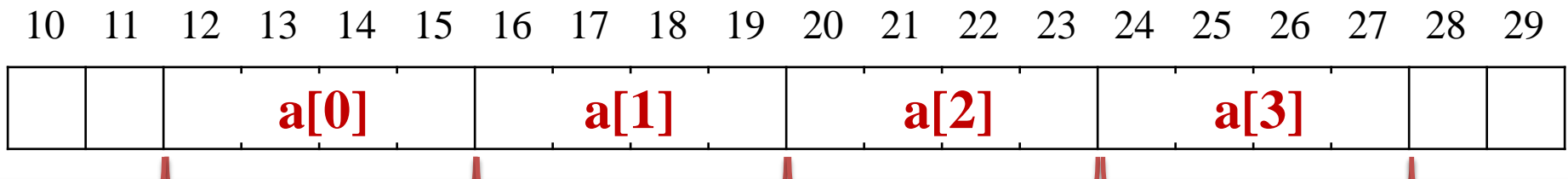
Выравнивание массивов

В связи с непрерывным расположением элементов массива, при их размещении в памяти достаточно обеспечить корректное выравнивание его первого элемента. Правильное выравнивание всех остальных элементов будет достигнуто автоматически.

Пример №1: неправильное расположение первого элемента приводит к неправильному расположению всех элементов.



Пример №2: правильное расположение первого элемента.

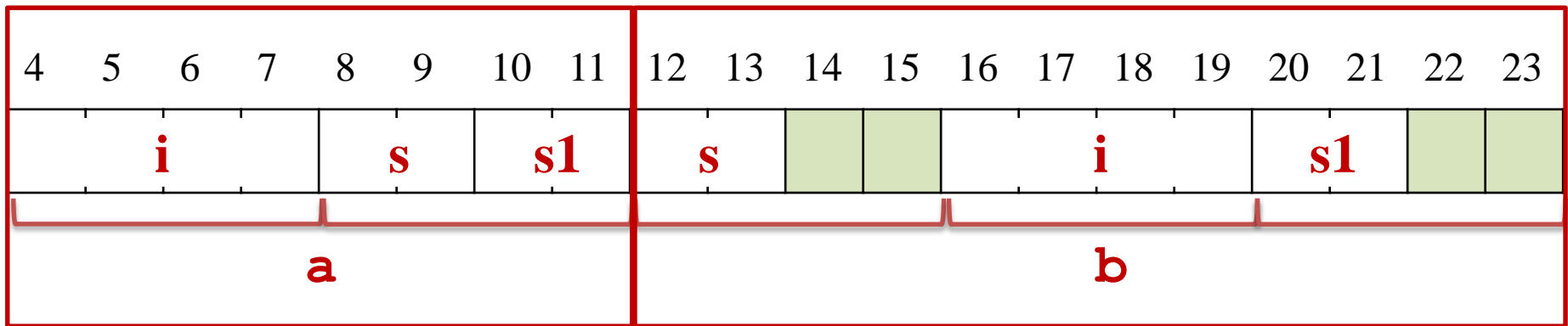




Выравнивание структур

```
struct t1{  
    int i;  
    short s;  
    short s1;  
} a;  
s_a = sizeof(a);
```

```
struct t1{  
    short s;  
    int i;  
    short s1;  
} b;  
s_b = sizeof(b);
```



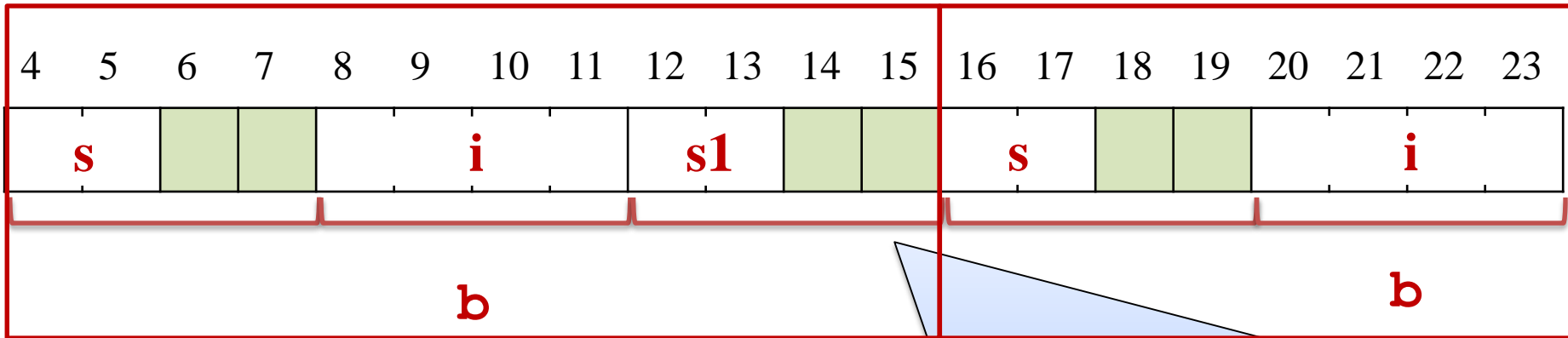
Правило выравнивания для структур:

Выравнивание структуры совпадает с наибольшим выравниванием среди ее элементов (в примере – int i).



Выравнивание массива структур

```
struct t1{  
    short s;  
    int i;  
    short s1;  
} b[10];
```



Данные байты не требуются для выравнивания индивидуальной структуры. Однако, они необходимы для обеспечения выравнивания набора структур, расположенных друг за другом (массива).



Переменная для компилятора

- Для **программиста** переменная – это ячейка памяти, в которой можно хранить значения соответствующего типа. Обращение к переменной производится по ее **имени**.
- Для компилятора и компоновщика переменная – это кортеж

```
#include <stdio.h>
int main()
{
    char c;
    short s;
    int i;
    float f;
    double d;
}
```

(type, name, address), описывающие блок памяти, расположенный по адресу **address**, в котором хранятся значения типа **type**. В программе переменная адресуется именем **name**.

```
(char, c, 10); (short, s, 12);  
(int, i, 16); (float, f, 20)
```

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

c		s			i		f			d
----------	--	----------	--	--	----------	--	----------	--	--	----------



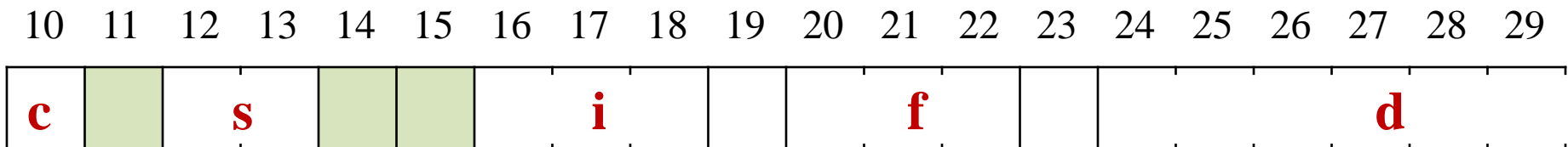
Переменная как объект языка

```
#include <stdio.h>
int main()
{
    char c;
    short s;
    int i;
    float f;
    double d;
}
```

При выполнении инструкции:

i = 5;

компилятором будет сгенерирована инструкция процессору, помещающая значение **5** по адресу **16**, ассоциированному с именем **i**.



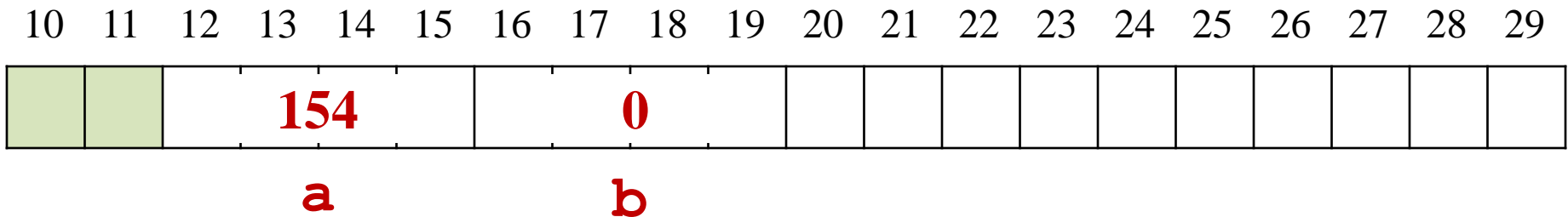
Таким образом, с каждой переменной в высокоуровневом языке программирования связано два числа:

- адрес (lvalue)
- значение (rvalue)



Контекст использования переменной

```
int a = 154, b = 0;
```



```
b = a;
```

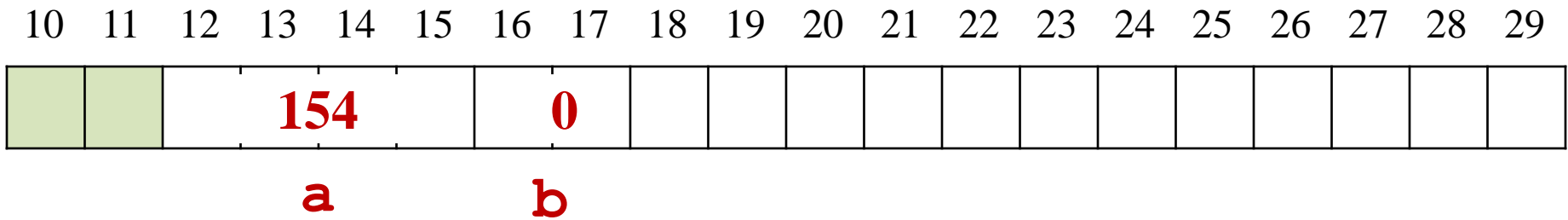
Выражение, расположенное справа от знака присваивания называется **rvalue** (right value). Переменные, входящие в это выражение рассматриваются в смысле "**значение**".

В приведенном выше примере из ячейки памяти **a**, начинающейся с **адреса 12** будет считано **sizeof(a)** байт, содержимое которых соответствует формату типа **int**.



Контекст использования переменной (2)

```
int a = 154;  
short b = 0;
```



```
b = a;
```

Выражение, расположенное слева от знака присваивания называется lvalue (left value). Переменные, входящие в это выражение рассматриваются в смысле "**адрес**".

В приведенном выше примере, результат выражения справа, будет преобразован к типу **short** b размещен в **sizeof(b)** байтах, начиная с **адреса 16**.



Указатели

- Указатели используются в языках высокого уровня для изменения умолчаний, связанных с контекстом использования переменных.
- Основной их функцией является возможность хранения адреса ячейки памяти.
- Для описания указателя на языке Си перед его именем необходимо указать знак '*':

```
int *i;
```

- Размер, необходимый для хранения указателя определяется аппаратурной платформой. На современных ПК: 4 байта (**x86**), 8 байт (**x86_64**).
- Фактический размер адреса не важен для программиста, т.к. компилятор берет на себя обработку этих деталей.



Указатели (2)

Рассмотрим следующий пример:

```
int *ptr;
```

- имя переменной-указателя с именем **ptr**;
- звездочка информирует компилятор что определяется указатель, т.е. выделяется ячейка памяти размером с адрес (одинаков для всех типов данных).
- тип `int` говорит о том что указатель будет хранить адрес целочисленного типа данных
- Говорят: "объявлен указатель на целое".

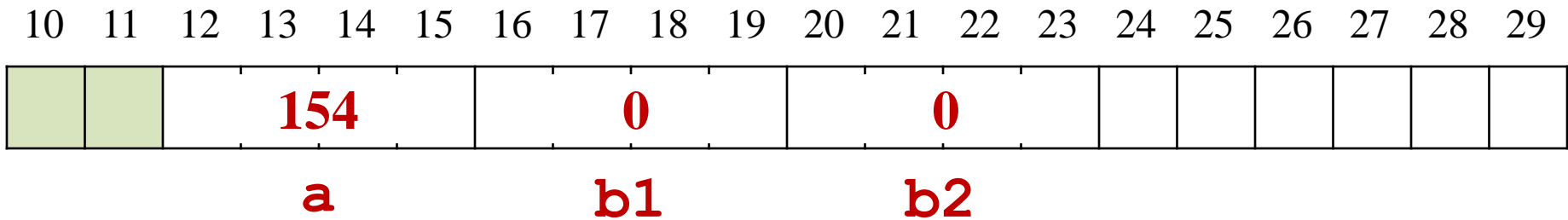


Операция взятия адреса

В языке Си существует операция взятия адреса, которая обозначается через амперсанд ("&").

Оператор "&" позволяет получить **адрес** (lvalue) переменной, если она располагается **справа** от знака присваивания, то есть по умолчанию трактуется как rvalue. Например:

```
int a = 154, b1 = 0, b2 = 0;
```



```
b1 = 10 + a;    // b1 == 164  
b2 = 10 + &a;   // b2 == 22
```



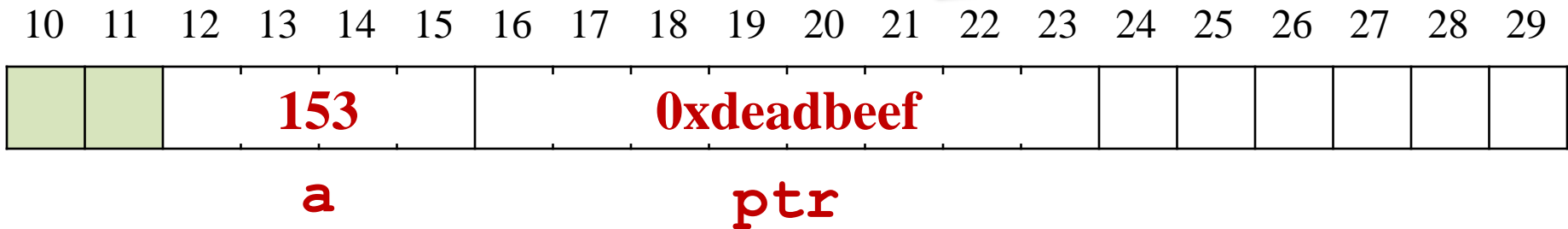
Настройка указателей

Операция взятия адреса используется для "настройки" указателя на реальную область памяти, например:

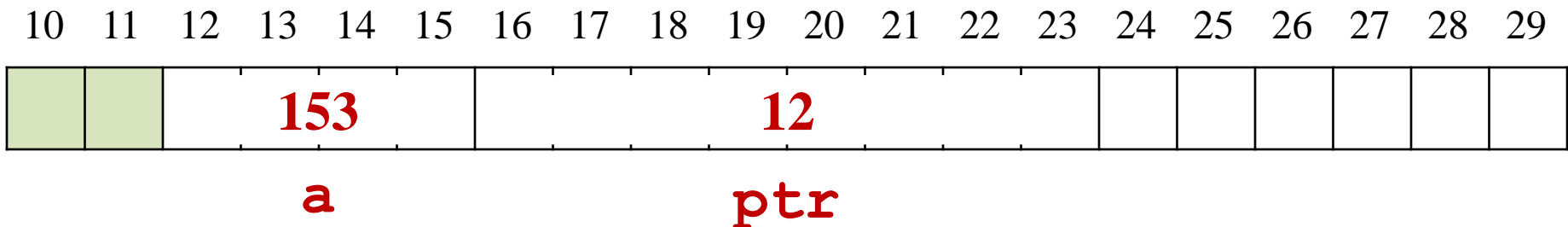
```
int a = 153, *ptr;
```

Hexspeak

<https://ru.wikipedia.org/wiki/Hexspeak>



```
ptr = a; // ptr == 12
```

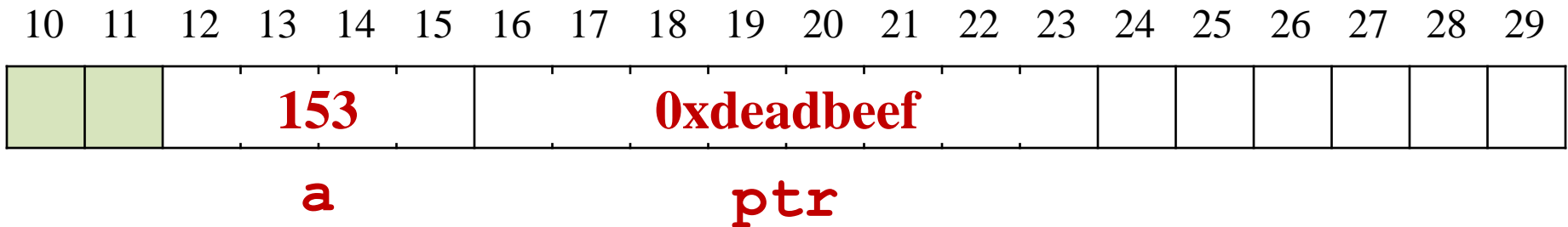




Типичная ошибка при настройка указателей

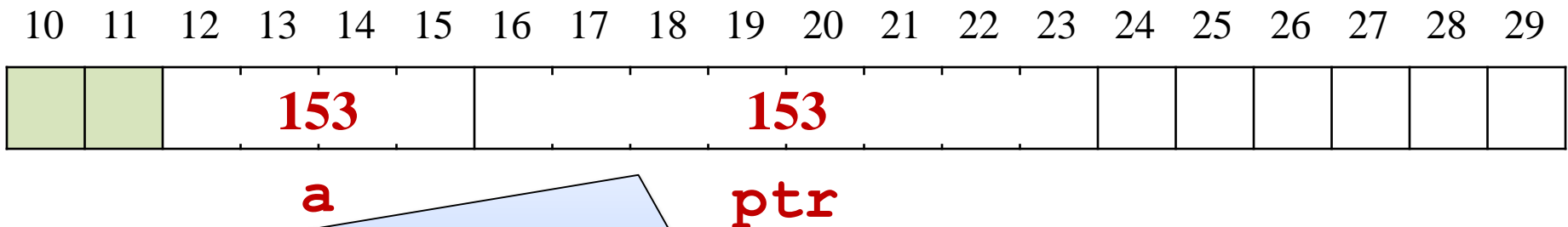
Операция взятия адреса используется для "настройки" указателя на реальную область памяти, например:

```
int a = 153, *ptr;
```



```
ptr = a; // ptr == 154 - ERROR!
```

154 →



ptr указывает на область памяти с адресом 153. Адрес противоречит правилу выравнивания. При корректном адресе обращение к такой ячейке приведет к нарушению целостности программы.(memory corruption).



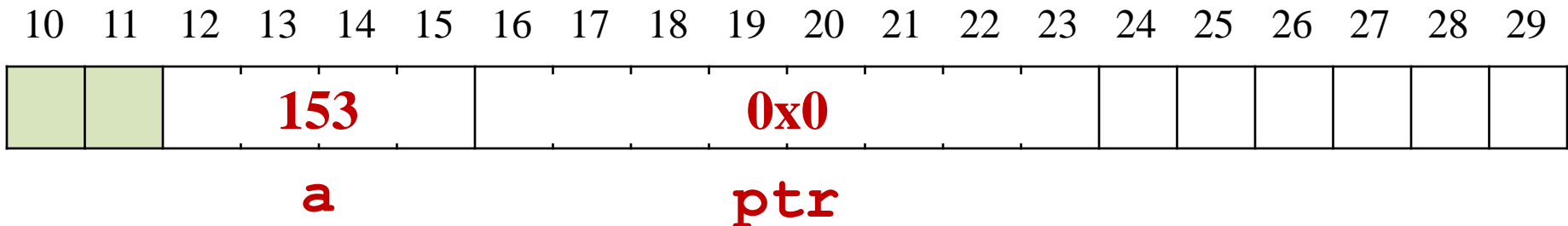
Нулевой указатель

Переменная-указатель может **не указывать ни на какую реальную ячейку памяти**:

- 1) при определении без инициализации;
- 2) в связи с алгоритмом выполнения программы.

Для того, чтобы явно задать значение указателя, не указывающего никуда, используется специальная константа `NULL`, которая в большинстве архитектур (но не во всех) равна 0. Обращение к нулевому указателю всегда приводит к аварийному завершению программы. Это помогает (при использовании отладчика) легко отслеживать некоторые ошибки использования указателей.

```
int a = 153, *ptr = NULL;
```





Контекст использования указателей

Для переменных-указателей в языке Си по умолчанию действуют те же правила определения используемого значения по контексту, что и для обычных переменных. Например:

```
int a = 153, *ptr = &a, b;  
b = ptr + 5; // 12 + 5 = 17
```

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
				153					12							17			
				a					ptr							b			

В приведенном примере указатель ptr используется справа от знака присваивания, поэтому он интерпретируется как "значение" и адрес, на который он указывает складывается с константой 5.

Приведенный пример не является корректным, т.к. значение указателя предполагается случайным.



```
int a = 153, *ptr = &a, b = 1;  
*ptr = b + 5; // положить 6, по адресу 12
```

В приведенном примере указатель `ptr` используется слева от знака присваивания, однако операция разыменования форсирует его трактовку как `rvalue`, поэтому он интерпретируется как "значение". Вместо того, чтобы изменять ячейку `ptr` будет выполнено считывание ее значения, которое будет интерпретироваться как адрес, по которому записывается результат.



Двойной указатель

Переменная-указатель, в свою очередь, имеет адрес. Что позволяет создавать ячейки памяти, предназначенные для хранения адресов ячеек с адресами, или двойных указателей.

Аналогичное рассуждение можно в свою очередь применить к двойным указателям, что позволит перейти к тройным переменным-указателям.

```
int a = 153, *p1 = &a, **p2 = &p1, ***p3 = &p2;
```

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
		6				12				16				20					
		a				p1				p2				p3					

***p3 == **p2 == *p1 == a == 153

**p3 == *p2 == p1 == &a == 12

*p3 == p2 == &p1 == 16

p3 == &p2 == 20



Задание на самостоятельную работу

10 11 **12** 13 14 15 **16** 17 18 19 **20** 21 22 23 **24** 25 26 27 28 29

		10	20	12	16	20
k1			k2	ptr1	ptr2	dptr

```
int k1 = 10, k2 = 20;
```

```
int *ptr1 = &k1, *ptr2 = &k2;
```

```
int **dptr = &ptr1;
```

```
**dptr = 80; // в какую ячейку записать?
```

```
dptr = &ptr2; // в какую ячейку записать?
```

```
**dptr = 100; // в какую ячейку записать?
```

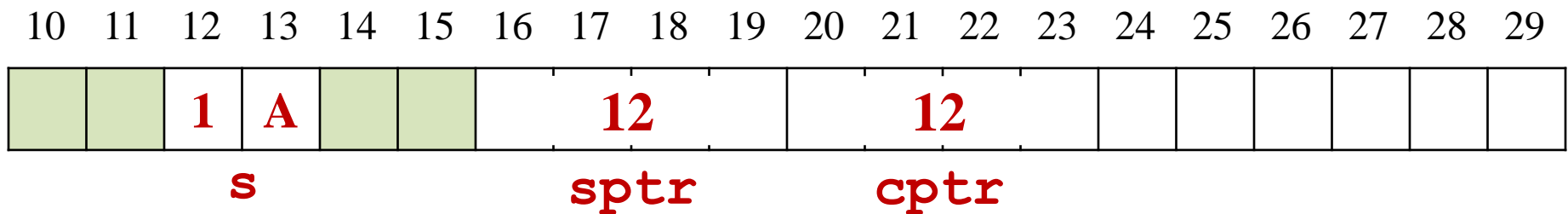
```
*dptr = ptr1; // в какую ячейку записать?
```

```
**dptr = 120; // в какую ячейку записать?
```



Тип указателя

```
unsigned short s = 0xA01, *sptr = &s;  
unsigned char *cptr = (unsigned char *)&s;
```



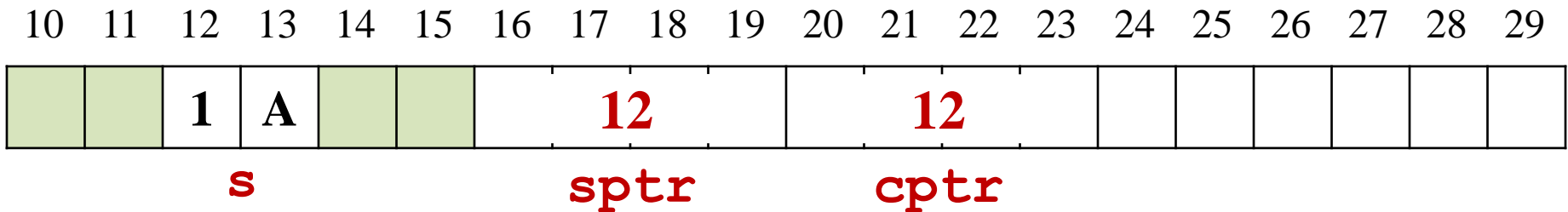
По определению **адрес** – это порядковый номер первого байта ячейки. Поэтому иногда возникает заблуждение о том, что тип указателя не нужен, так как адрес имеет одинаковый формат. Однако, в процессе присваивания необходима информация не только о первом байте ячейки, но и о ее размере. В приведенном ниже примере результат присваивания не будет корректным:

```
*cptr = 2;
```



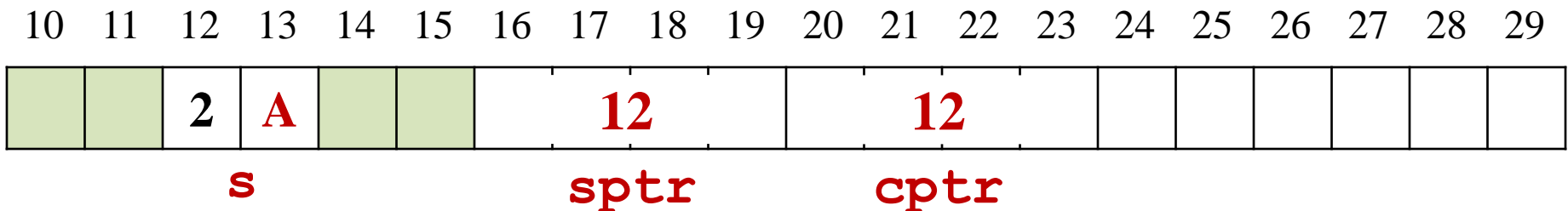
Тип указателя (2)

```
unsigned short s = 0xA01, *sptr = &s;  
unsigned char *cptr = (unsigned char *)&s;
```



В приведенном ниже примере результат присваивания не будет корректным:

```
*cptr = 2; // s == A02 вместо 2!
```





Допустимые операции над указателями

```
int j, *ptr1, *ptr2, **dptr;
```

1	Операция присваивания:	<code>ptr1 = &j;</code>
2	Операция взятия адреса:	<code>dptr = &ptr1;</code>
3	Операция разыменования:	<code>j = *ptr2;</code>
4	Сложение с целым:	<code>ptr2 = ptr1 + j;</code>
5	Разность указателей:	<code>j = ptr1 - ptr2;</code>
6	Операция индексации:	<code>ptr1[j] = 10;</code>

Адресная арифметика



Связь массивов и указателей

Имя массива является **указателем-константой на его первый элемент**. Например:

```
int a[4] = {1, 2, 3, 4}, *ptr = a;  
// в программе имя массива a заменяется на значение  
// указателя на его первый элемент 12.  
// Ячейки a не существует!
```

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
			1			2			3			4						12					
			a[0]			a[1]			a[2]			a[3]						ptr					

$a[i] \sim ptr[i] \sim *(ptr + i)$

Адресная арифметика.
Сложение с целым



Адресная арифметика [сложение с целым]

```
int a[4] = {1, 2, 3, 4}, *ptr = a;
```

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
			1				2				3				4				12				
			a[0]				a[1]				a[2]				a[3]				ptr				

```
ptr == 12  
ptr + 1 == 16 // а не 13!  
ptr + 2 == 20 // а не 14!  
ptr + 3 == 24 // а не 15!
```

При сложении указателя с целым числом указатель сдвигается не на 1-цу, а на размер элемента, на который он указывает (в данном примере это `int`, `sizeof(int) == 4`)



Адресная арифметика [разность указателей]

```
int a[4] = {1, 2, 3, 4};  
int *ptr1 = a, *ptr2 = &a[2];
```

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
		1				2			3			4					12						
		a[0]				a[1]			a[2]			a[3]					ptr						

```
ptr2 - ptr1 = 2 // а не 20 - 12 = 8
```

При вычитании указателей результат указывает на количество элементов, расположенных между ними, а не расстояние в байтах!



Адресная арифметика [индексация]

```
int a[4] = {1, 2, 3, 4}, *ptr = a;
```

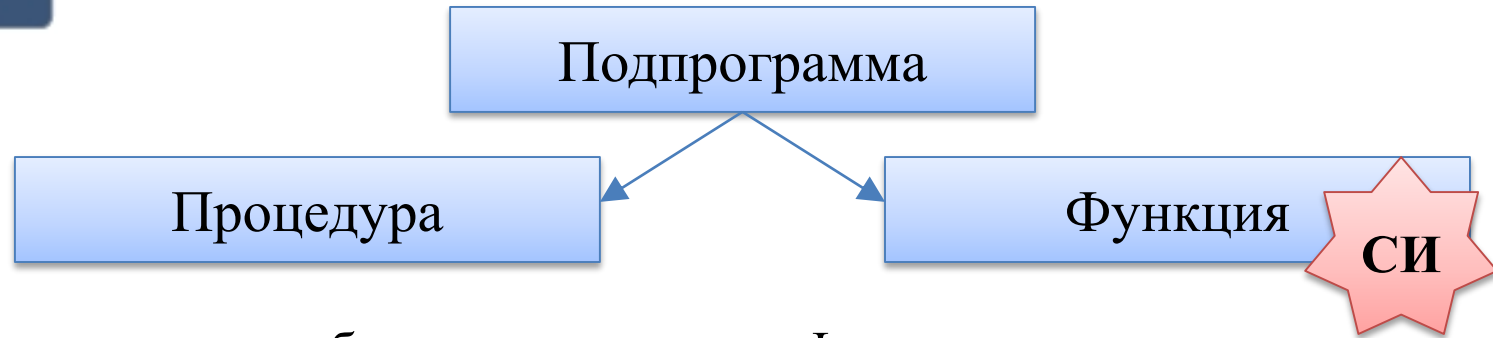
11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
		1				2				3			4					12					
		a[0]				a[1]				a[2]			a[3]					ptr					

```
* (ptr + 1) = 10; // * (12 + 1*4) = * (16)  
* (ptr + 2) = 11; // * (12 + 2*4) = * (20)  
* (ptr + 3) = 12; // * (12 + 3*4) = * (24)
```

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
		1				10				11			12					12					
		a[0]				a[1]				a[2]			a[3]					ptr					



Процедуры и функции



Процедура – это набор операторов, реализующих *параметризованные* вычисления, которые активизируются отдельными операторами вызова.

Процедуры определяют новые *операторы* языка, например сортировку элементов массива.

Процедура вырабатывает результат через глобальные переменные или формальные параметры, позволяющие передавать данные в вызывающий модуль.

Функции семантически моделируют математические функции, где не допускается изменение их параметров или ячеек, определенных вне функции.

Функции вызываются через указание ее имени и соответствующих фактических параметров. *Значение, вычисленное функцией заменяет собой ее вызов!*

Для реализации процедур в языке Си используются указатели!

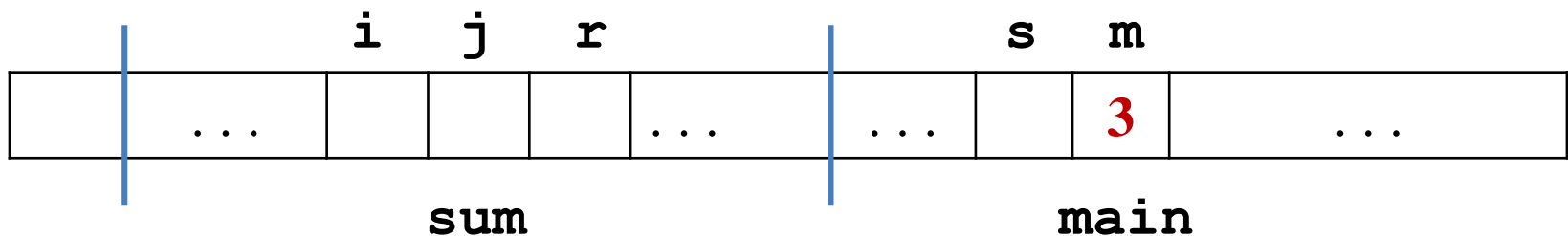


Передача параметров по значению (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
int sum(int i,int j)
{
    int r = i + j;
    return r;
}
```

```
int main(){
    ➡ int s, m=3;
      s = sum(m, 5);
}
```





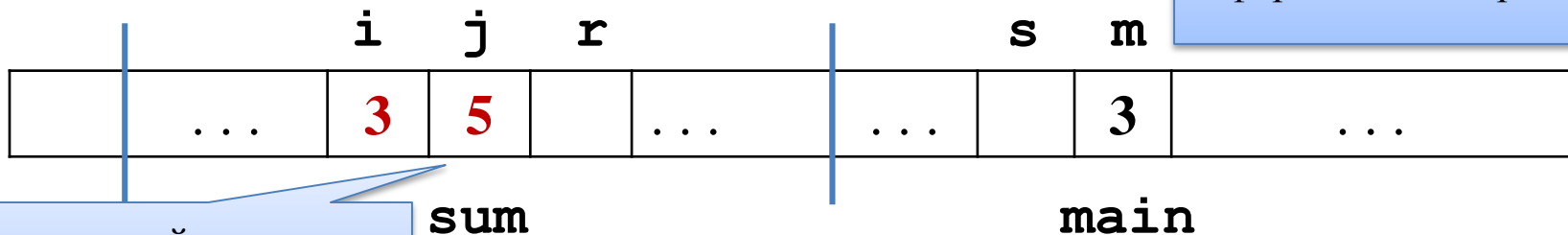
Передача параметров по значению (вызов sum)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
int sum(int i, int j)
{
    int r = i + j;
    return r;
}
```

```
int main() {
    int s, m = 3;
    s = sum(m, 5);
}
```

Значение фактического параметра *m* копируется в формальный параметр *i*



Фактический параметр-константа 5 копируется в формальный параметр *j*

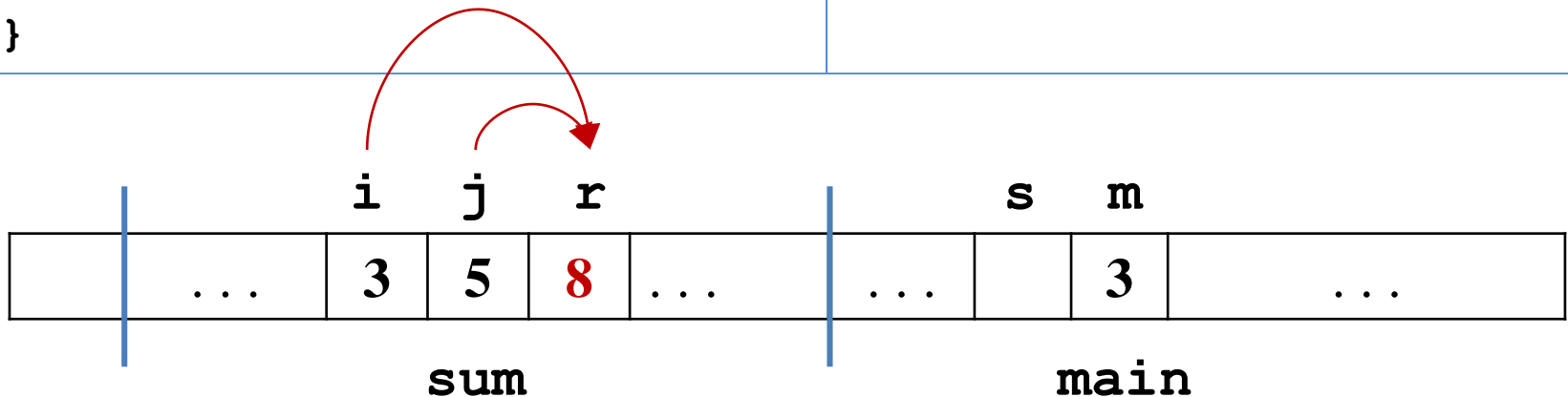


Передача параметров по значению (выполнение тела функции sum)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
int sum(int i,int j)
{
  int r = i + j;
  return r;
}
```

```
int main(){
    int s, m=3;
    s = sum(m, 5);
}
```



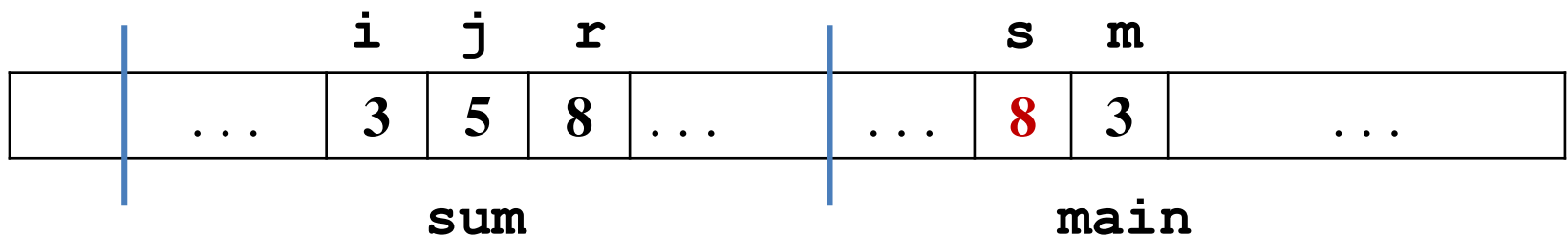


Передача параметров по значению (возврат значения из sum в main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
int sum(int i,int j)
{
    int r = i + j;
    return r;
}
```

```
int main(){
    int s, m=3;
    s = sum(m, 5);
}
```



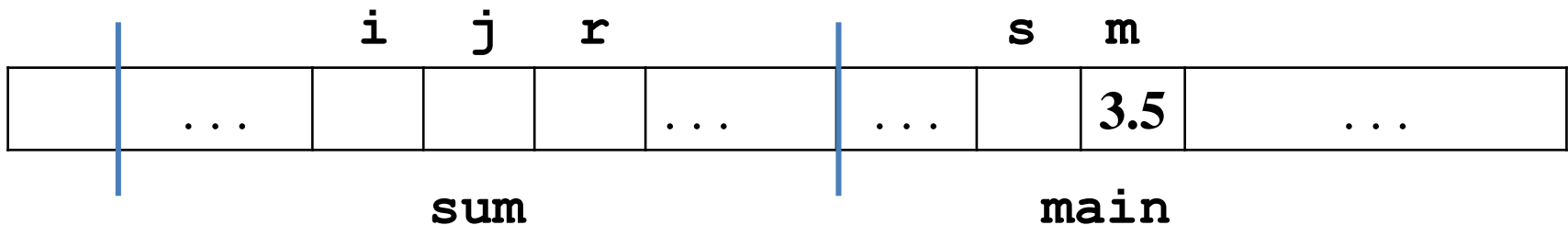


Приведение типов фактических параметров к формальным (вызов main)

При вызове функции значения фактических параметров **приводятся** к типу данного формального параметра.

```
int sum(int i,int j)
{
    int r = i + j;
    return r;
}
```

```
int main(){
    ➔ float s, m=3.5;
      s = sum(m, 5.5);
}
```



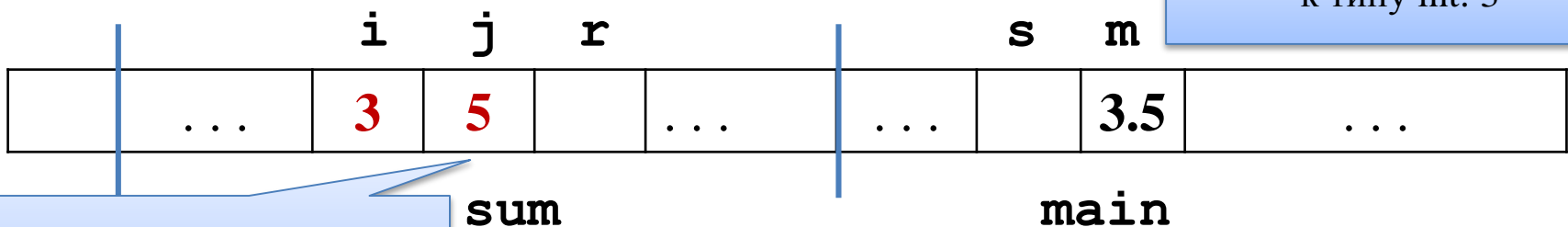


Приведение типов фактических параметров к формальным (вызов sum)

При вызове функции значения фактических параметров **приводятся** к типу данного формального параметра.

```
int sum(int i,int j)
{
    int r = i + j;
    return r;
}
```

```
int main(){
    float s, m=3.5;
    → s = sum(m, 5.5);
}
```



(float)5.5 приводится к
типу int: 5

float m = 5.5 приводится
к типу int: 3

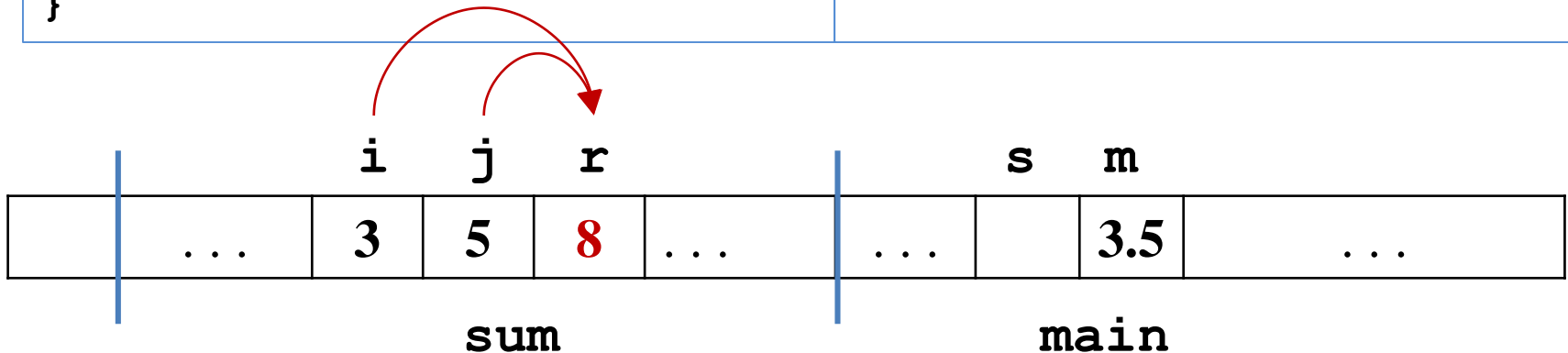


Приведение типов фактических параметров к формальным (тело sum)

При вызове функции значения фактических параметров **приводятся** к типу данного формального параметра.

```
int sum(int i,int j)
{
  int r = i + j;
  return r;
}
```

```
int main(){
  float s, m=3.5;
  s = sum(m, 5.5);
}
```



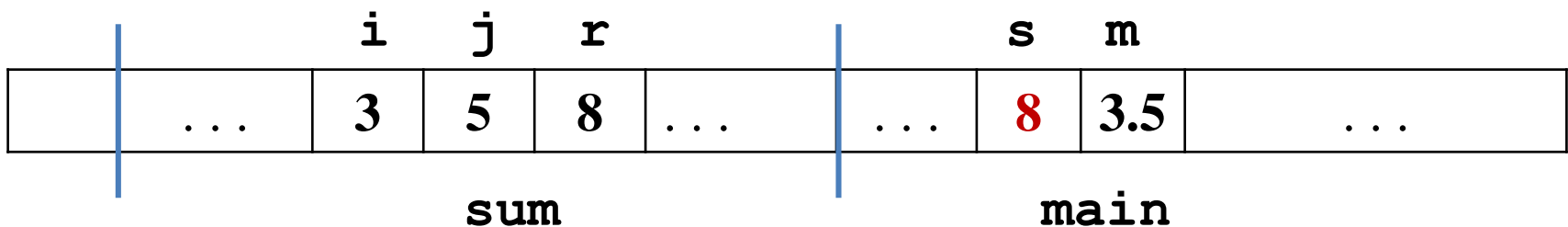


Приведение типов фактических параметров к формальным (возврат из sum)

При вызове функции значения фактических параметров **приводятся** к типу данного формального параметра.

```
int sum(int i,int j)
{
    int r = i + j;
    return r;
}
```

```
int main(){
    float s, m=3.5;
    s = sum(m, 5.5);
}
```





Реализация процедуры в языке СИ


- Процедура предполагает параметризацию кода, результаты которого возвращаются через аргументы процедуры.
- Ключевым моментом является то, что изменения формальных параметров *не отражаются* на фактических (см. слайды ниже). Это связано с тем, что в качестве фактического параметра могут передаваться константы, в этом случае обратная связь формального и фактического параметра невозможна.
- Для организации возврата результатов через аргументы функции (а не через оператор `return`) требуется использование указателей.
- Изменения массивов возвращаются в вызывающую подпрограмму, т.к. массивы организованы на базе неявных указателей: операция `a[x]` индексации работает с указателями.

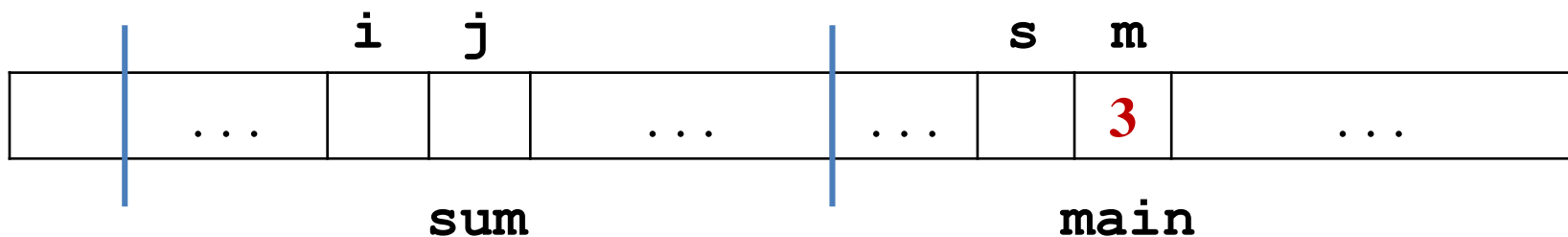


Пример **неправильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int i,int j)
{
    i = i + j;
}
```

```
int main(){
     int m=3;
    sum(m, 5);
    ...
}
```





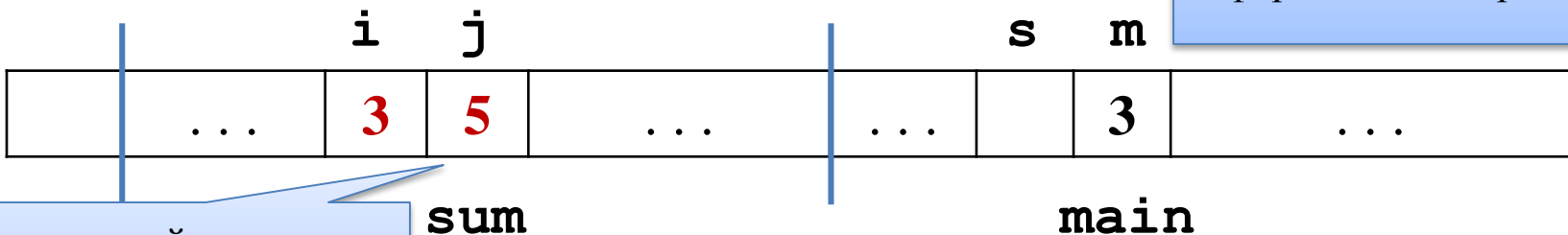
Пример **неправильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int i, int j)
{
    i = i + j;
}
```

```
int main() {
    int m=3;
    → sum(m, 5);
}
```

Значение фактического параметра *m* копируется в формальный параметр *i*




Фактический параметр-константа 5 копируется в формальный параметр *j*



Пример **неправильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int i, int j)
{
   i = i + j;
}
```

```
int main() {
  int m=3;
  sum(m, 5);
  ...
}
```



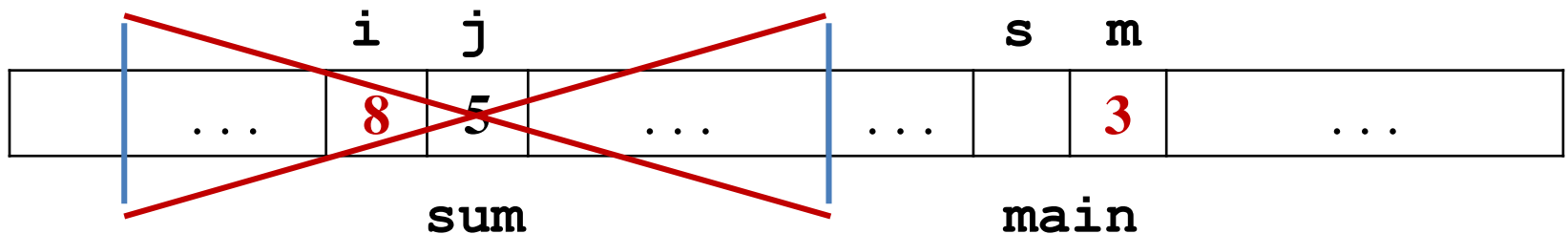


Пример **неправильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int i,int j)
{
    i = i + j;
}
```

```
int main(){
    int m=3;
    sum(m, 5);
    ...
}
```



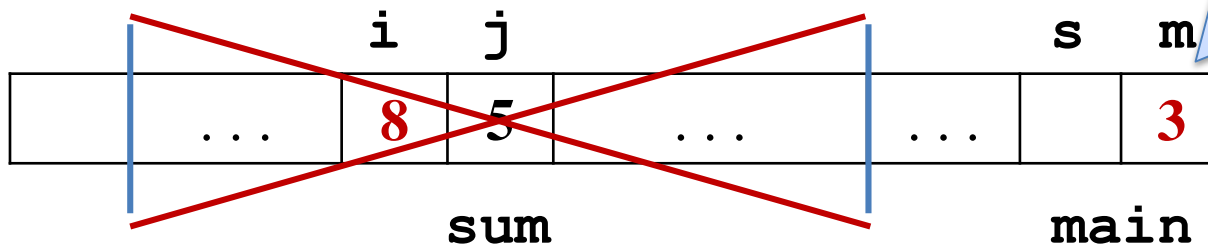


Пример **неправильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int i, int j)
{
    i = i + j;
}
```

```
int main() {
    int m=3;
    sum(m, 5);
    ...
}
```



После завершения sum изменения локальной переменной i, являющейся формальным параметром, недоступны в функции main в фактическом параметре m



Указатели

- Указатель: специальным образом описанная ячейка памяти, которая хранит адрес типизированной ячейки. Самостоятельного значения указатель не имеет.
- Для получения адреса ячейки памяти (переменной) используется операция "амперсанд": &.
- Описание указателя: перед именем переменной ставится "звездочка":

```
int *x; float d, *p = &d;
```

d – ячейка
p – указатель на d

- Если в ячейке-указателе находится корректный адрес ячейки базового типа, то значение ячейки базового типа можно читать и изменять через указатель, используя операцию "разыменования", которая обозначается символом "звездочка":

```
d = 10 и *p = 10
```

Результат
одинаков



Указатели (пример)

```
float d = 10, *p = &d;
```

d

p

...	10	0xF1	...
-----	----	------	-----

0xF1

Адрес ячейки d – порядковый номер первого байта этой ячейки от начала памяти программы

p = 20;

d

p

...	10	20	...
-----	----	----	-----

Неверно! Изменено содержимое p!
Что находится в 20-м байте программы???

0xF1

***p = 20;**

d

p

...	20	0xF1	...
-----	----	------	-----

Верно! Изменено содержимое ячейки с адресом 0xF1, на которую ссылается p!

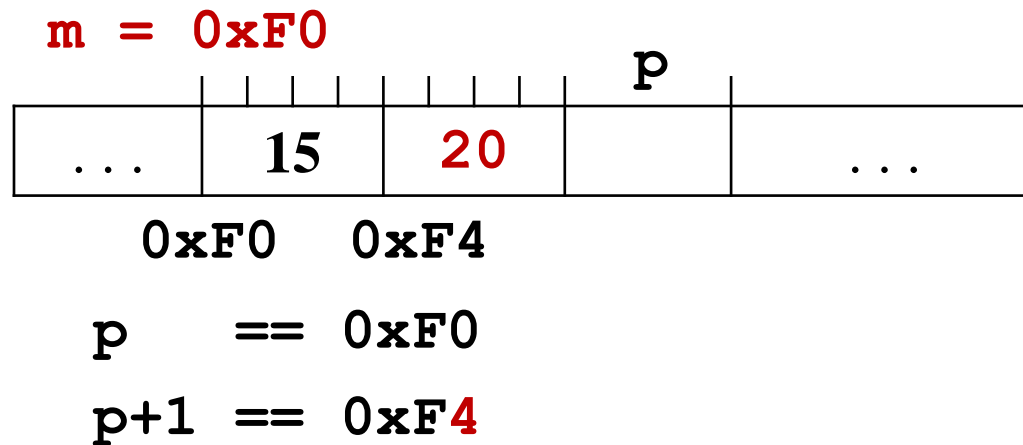
0xF1



Адресная арифметика (сложение с целым)

```
int m[2], *p = m;  
*p = 15;  
*(p + 1) = 20; // ~ p[1]
```

Имя массива – **УКАЗАТЕЛЬ-КОНСТАНТА** на его первый элемент



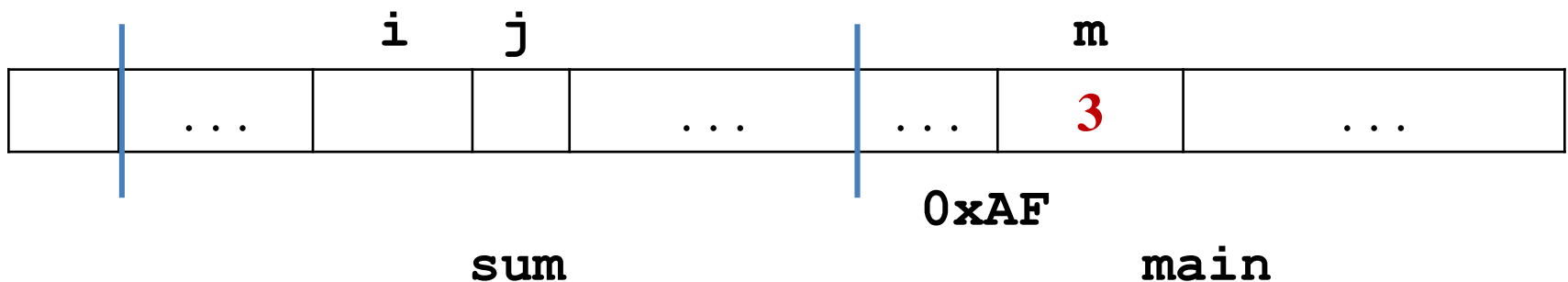


Пример **правильной** процедуры (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int *i,int j)
{
    *i = *i + j;
}
```

```
int main(){
    ➡ int m=3;
      sum(&m, 5);
      ...
}
```





Пример **правильной** процедуры (вызов функции sum)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int *i, int j)
{
    *i = *i + j;
}
```

```
int main() {
    int m=3;
    → sum(&m, 5);
}
```

Адрес переменной *m* используется для инициализации формального параметра *i*



Фактический параметр-константа 5 копируется в формальный параметр *j*

sum

0xAF

main



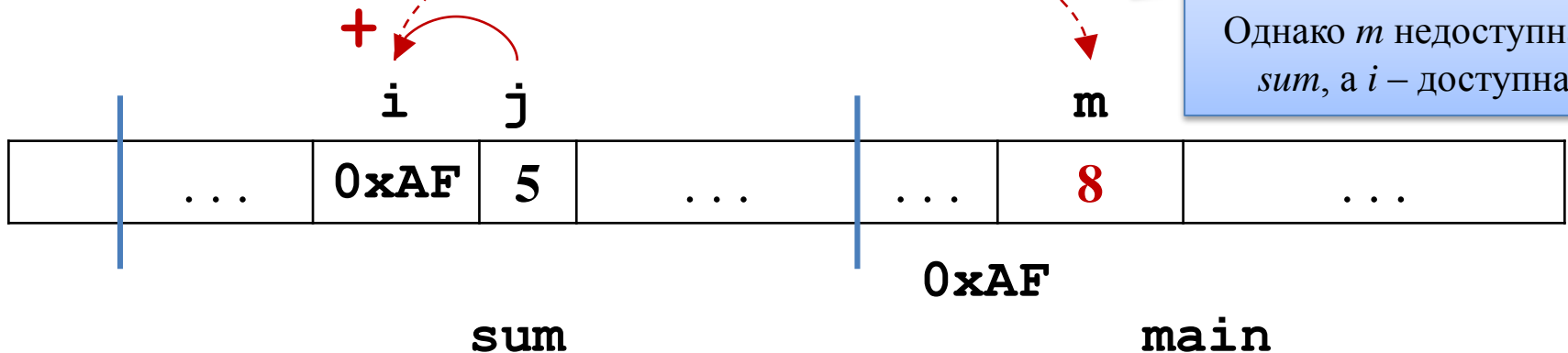
Пример **правильной** процедуры (выполнение тела `sum`)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int *i, int j)
{
    *i = *i + j;
}
```

```
int main() {
    int m=3;
    sum(&m, 5);
}
```

В выражении значение `*i` является синонимом `m`. Однако `m` недоступна в `sum`, а `i` – доступна.





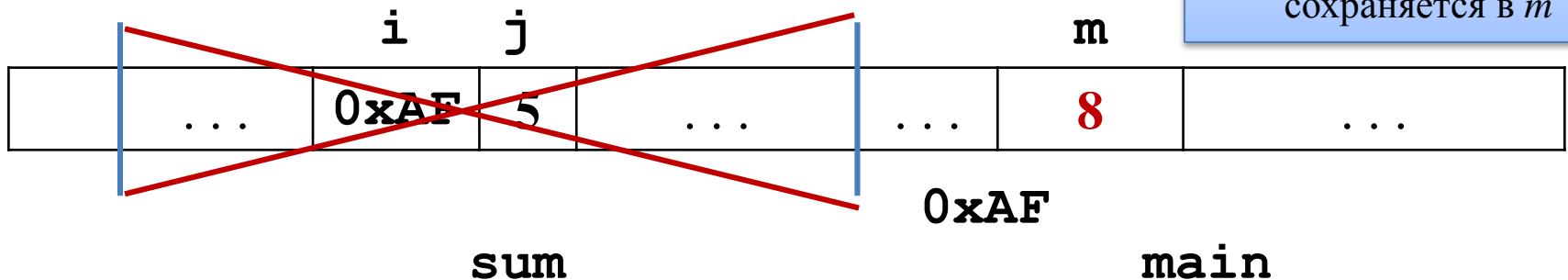
Пример **правильной** процедуры (завершение `sum`, переход в `main`)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void sum(int *i, int j)
{
    *i = *i + j;
}
```

```
int main() {
    int m=3;
    sum(&m, 5);
}
```

После уничтожения
локальных переменных
функции *sum* результат
сохраняется в *m*



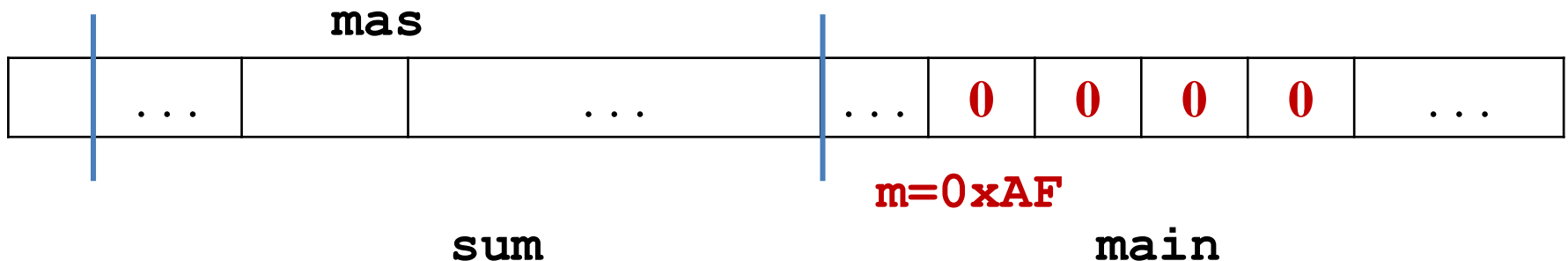


Процедура обработки массива (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void func(int mas[])  
{  
    mas[3] = 5;  
}
```

```
int main(){  
    ➡ int m[4] = {0};  
    func(m);  
    ...  
}
```





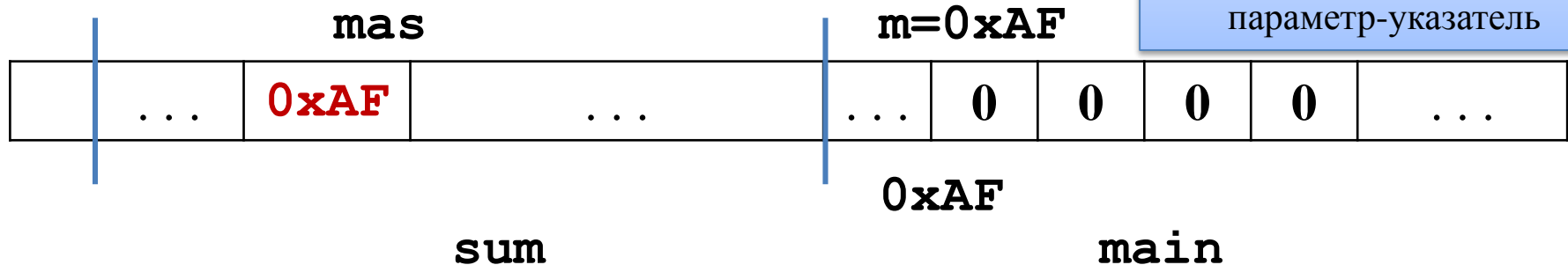
Процедура обработки массива (запуск функции main)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void func(int mas[])  
{  
    mas[3] = 5;  
}
```

```
int main(){  
    int m[4] = {0};  
    func(m);  
}
```

Имя массива – указатель-константа на его первый элемент. При вызове он копируется в формальный параметр-указатель





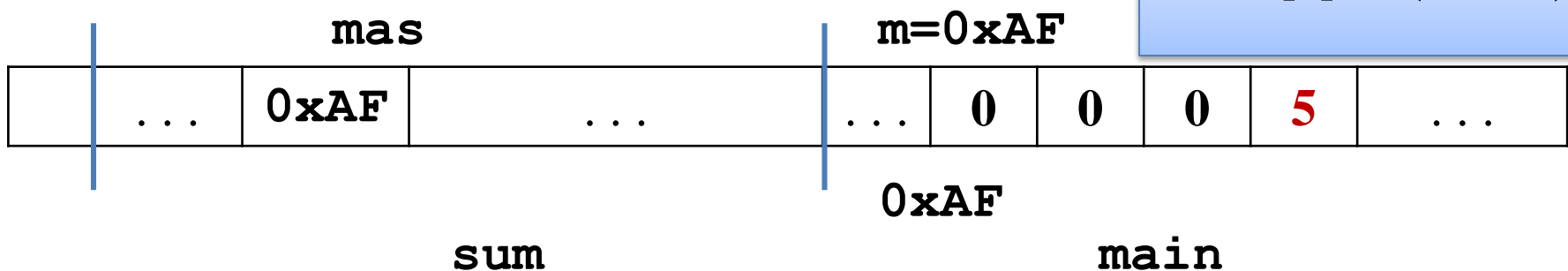
Процедура обработки массива (выполнение тела sum)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void func(int mas[])  
{  
    → mas[3] = 5;  
}
```

```
int main(){  
    int m[4] = {0};  
    func(m);  
    ...  
}
```

mas[3] ~ *(mas+3)





Процедура обработки массива (завершение `sum`, переход в `main`)

Формальные параметры функции – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
void func(int mas[])  
{  
    mas[3] = 5;  
}
```

```
int main(){  
    int m[4] = {0};  
    func(m);  
}
```

После уничтожения локальных переменных функции *sum* результат сохраняется в *m*

