ФГОБУ ВПО "СибГУТИ"
Кафедра вычислительных систем

ЯЗЫКИ ПРОГРАММИРОВАНИЯ / ПРОГРАММИРОВАНИЕ

УКАЗАТЕЛИ

Преподаватель:

Доцент Кафедры ВС, к.т.н.

Поляков Артем Юрьевич

Переменная

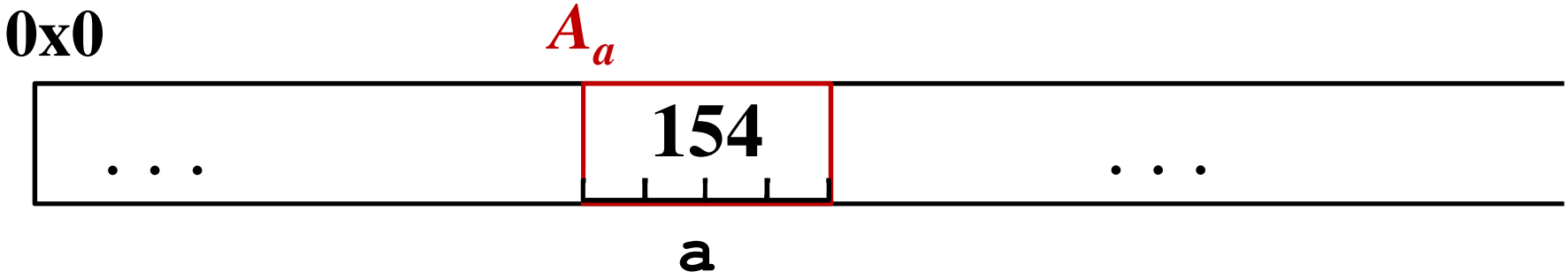
Компьютерная память хранит большой объем разнообразной информации. Для доступа к конкретной ее части *необходим адрес*.

Переменная в программе – объект, имеющий имя, содержимое которого может меняться.

Переменная для компилятора

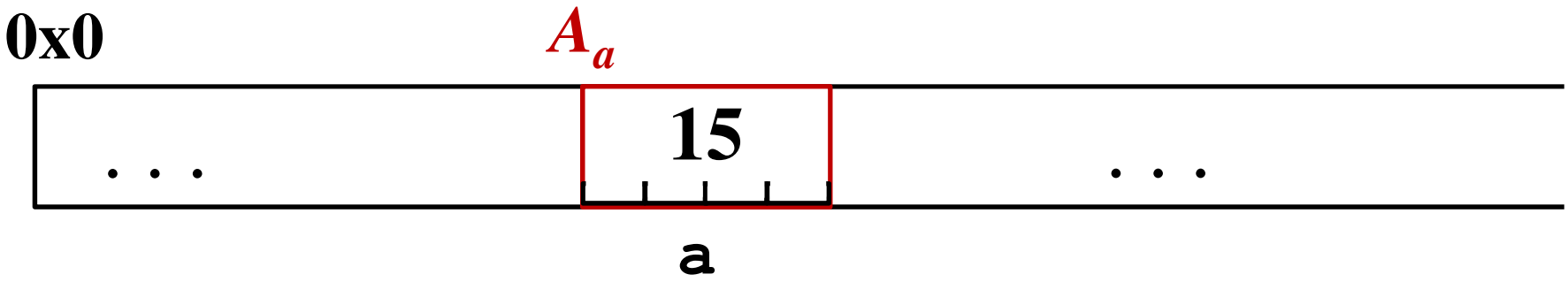
- Для компилятора и компоновщика переменная – это блок памяти, хранящий значение переменной, обратиться к которому можно по имени переменной.
- Размер блока определяется типом переменной. Например размер типа int на современных ПК – 4 байта.
- При определении переменной компилятору предоставляется информация о типе и имени переменной и, возможно, о ее значении:

```
int a = 154;
```



Выполнение программы

```
int a = 154;
```



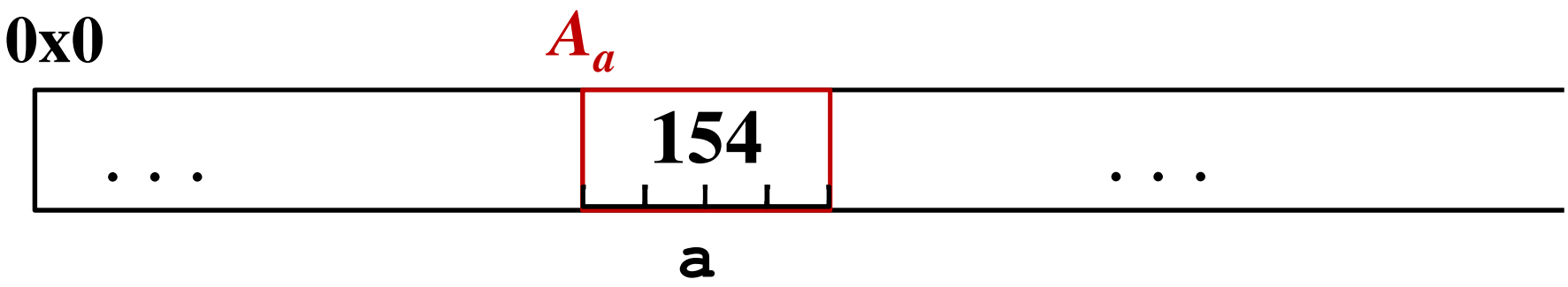
При выполнении инструкции:

```
a = 15;
```

компилятором будет сгенерирована инструкция процессору, помещающая значение 15 по адресу A_a, ассоциированному с символом *a*.

Переменная как объект языка программирования

```
int a = 154;
```

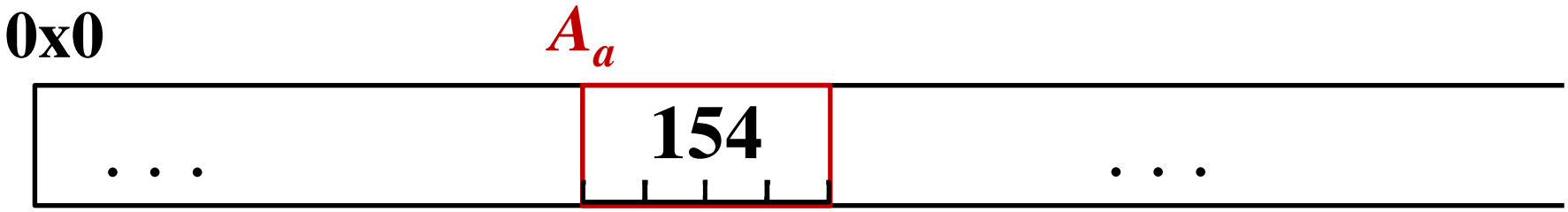


Таким образом, с каждой переменной в высокоуровневом языке программирования связано два числа:

- **адрес переменной (lvalue)**
- **значение переменной (rvalue)**

Контекст использования переменной (rvalue – справа от знака присваивания)

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

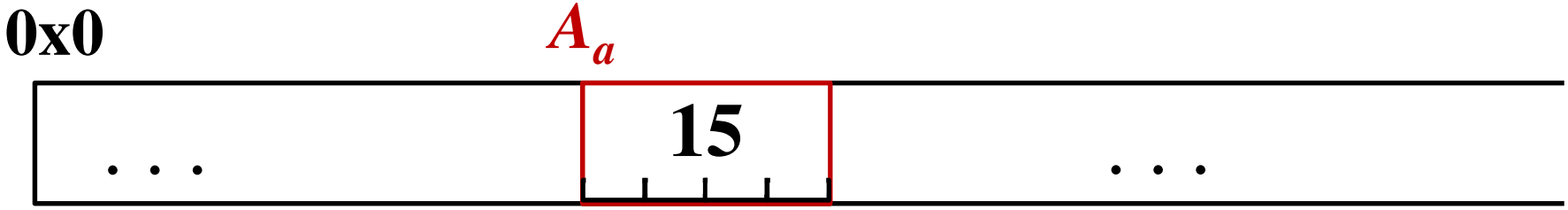



```
b = a;
```
<-- *a* рассматривается как rvalue, т.е. 15

Если переменная расположена справа от знака присваивания она рассматривается в смысле "**значение переменной**", т.е.:

из ячейки памяти, начинающейся с адреса A_a будет взято sizeof(*a*) байт, которые будут интерпретироваться как значение типа int.

Контекст использования переменной (lvalue – слева от знака присваивания)

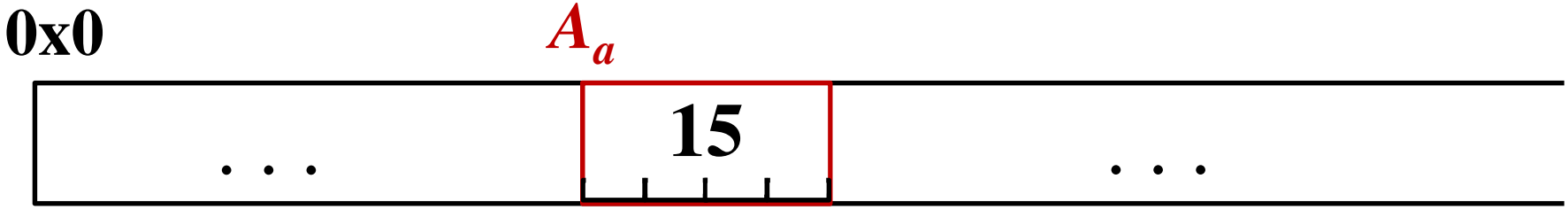


`a = 15;` <-- *a* рассматривается как lvalue, т.е. A_a

Если переменная расположена слева от знака присваивания она рассматривается в смысле "**адрес переменной**", т.е.:

в ячейку памяти, начинающуюся с адреса A_a будет записано sizeof(*a*) байт, которые будут преобразованы к типу int и определяются выражением справа от знака присваивания.

Комбинации контекста и значения, связанного с переменной




```
int a = 5, b = 10;
a = b;
```

Не имеет смысла,
как и:
7 = 5 или 7 = b

	a	*b*
rvalue	нельзя	контекст
lvalue	контекст	**?**

Указатели (2)

Указатели используются в языках высокого уровня для изменения умолчаний, связанных с контекстом использования переменных.

Основной их функцией является возможность хранения адреса ячейки памяти.

Указатели (2)

- 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 (3)

Рассмотрим следующий пример:

```
int *ptr;
```

- имя переменной-указателя (в дальнейшем просто указатель) – 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, несмотря на то, что. по умолчанию переменная будет рассматриваться как lvalue:

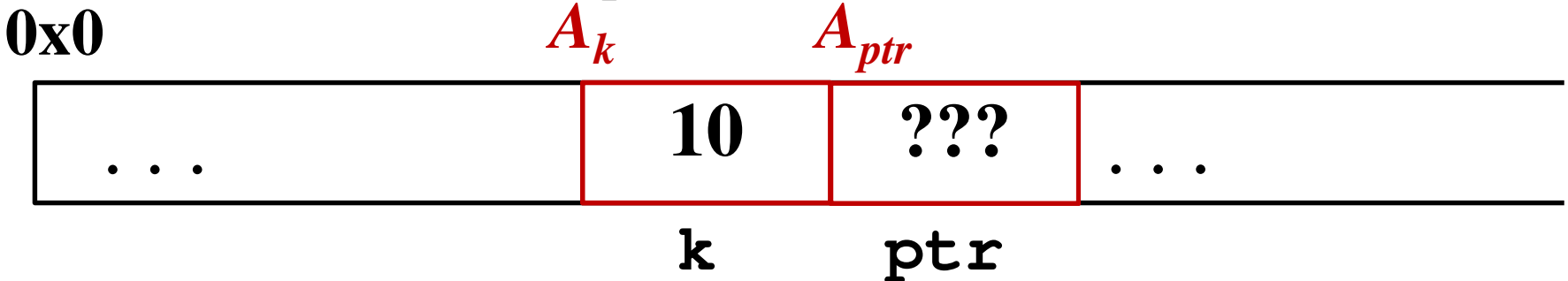
```
int a = 10, b = 15, x;
... = &a - b + 8
```

В указанном примере переменная *a* рассматривается как адрес!

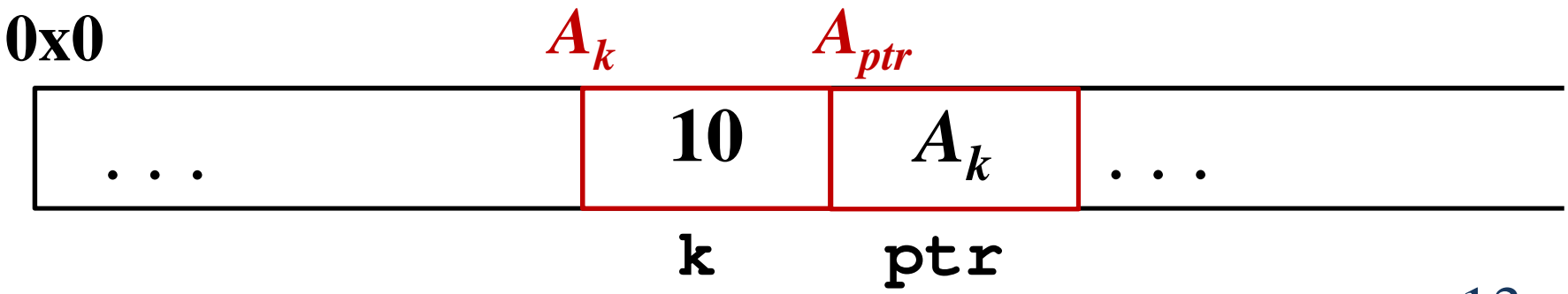
Настройка указателей

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

```
int k = 10, *ptr;
```




```
ptr = &k;
```



Нулевой указатель

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

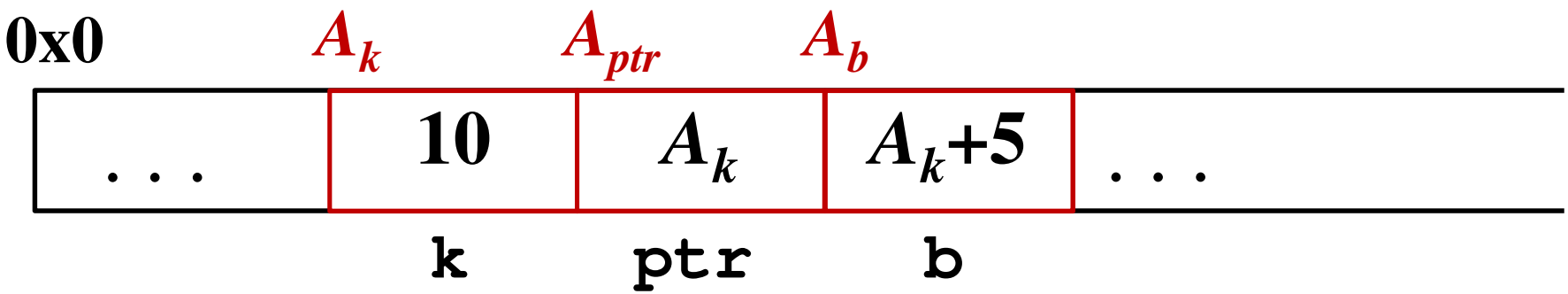
1) при определении без инициализации;
2) в связи с алгоритмом выполнения программы.

Для того, чтобы явно задать значение указателя, не указывающего никуда, используется специальная константа NULL, которая в большинстве архитектур (но не во всех) равна 0.

Контекст использования указателей

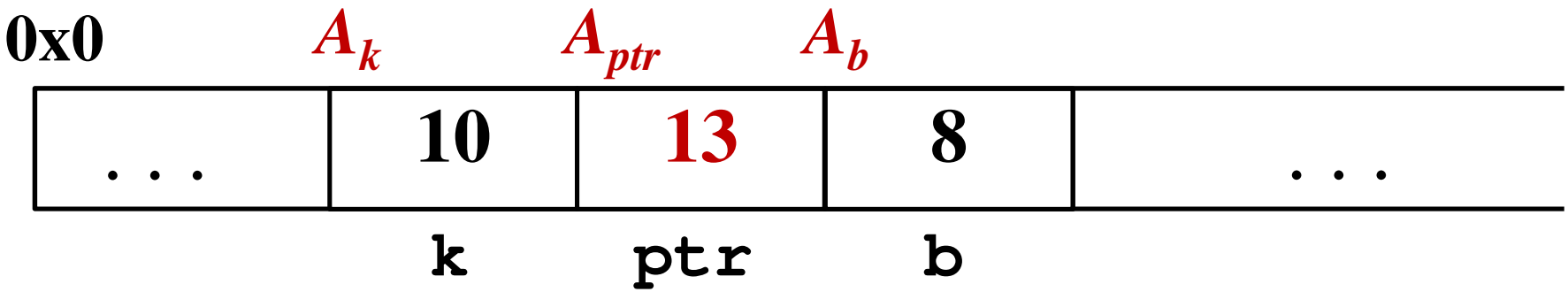
Для переменных-указателей в языке Си по умолчанию действуют те же правила определения используемого значения по контексту, что и для обычных переменных. Например:

```
int k = 10, *ptr = &k, b;
b = ptr + 5;
```



Контекст использования указателей (2)

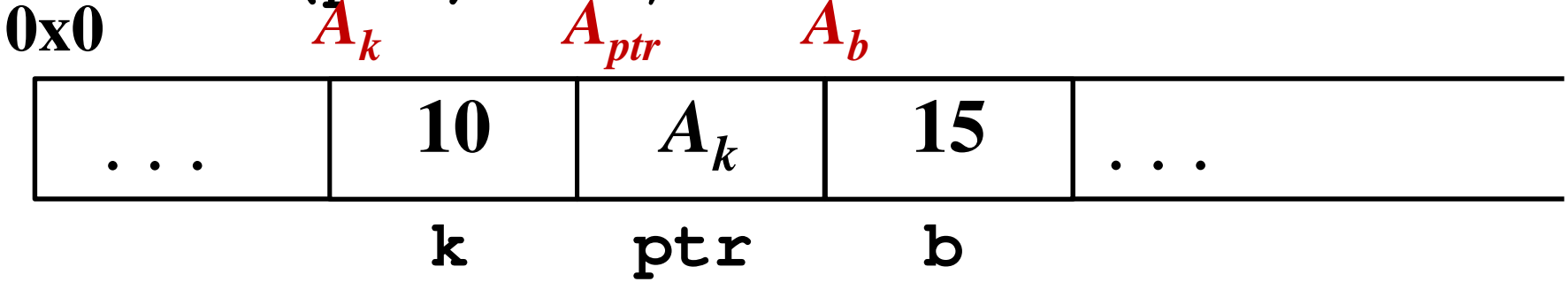
```
int k = 10, *ptr = &k, b = 8;
ptr = b + 5;
```



Операция разыменования

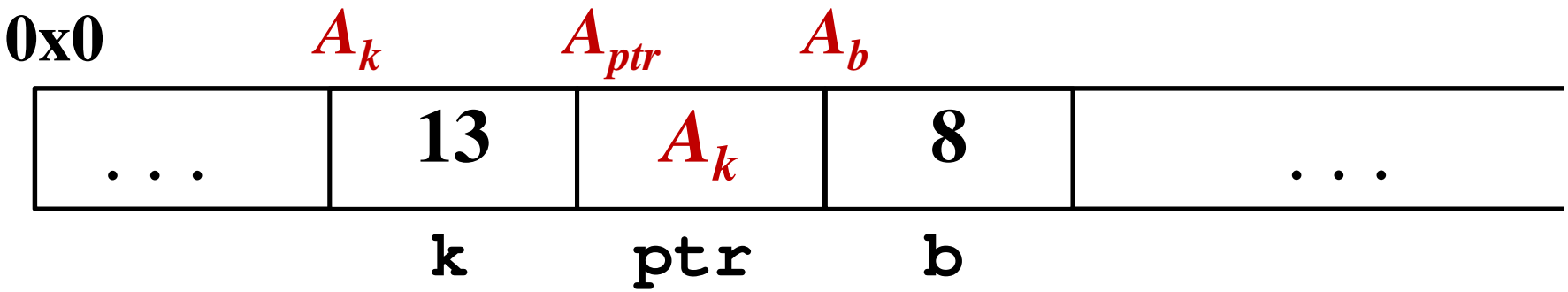
Для изменения стандартного распознавания переменной-указателя в выражениях используется операция разыменования, которая обозначается через "звездочку" ("*"):

```
int k = 10, *ptr = &k, b;
b = k + 5;
b = *(ptr) + 5;
```



Операция разыменования (2)

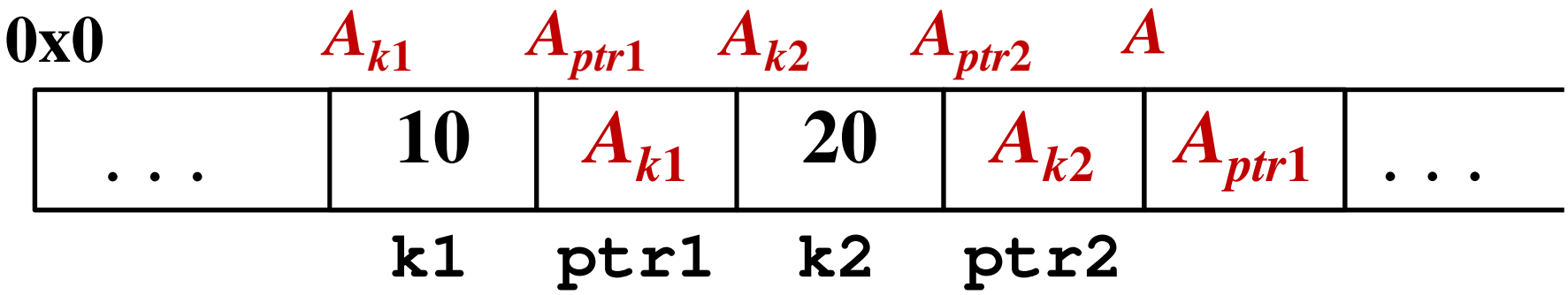
```
int k = 10, *ptr = &k, b = 8;
*ptr = b + 5;
```



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.



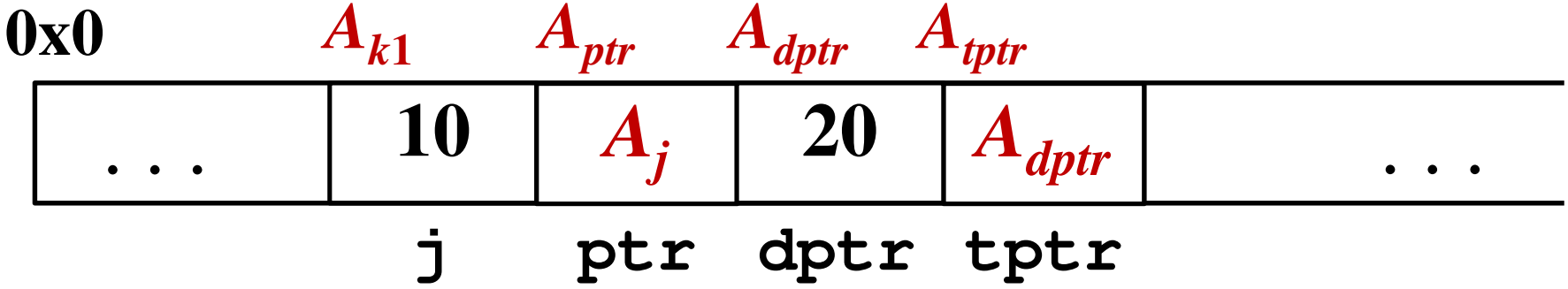
Двойной указатель

В общем существует следующее правило:

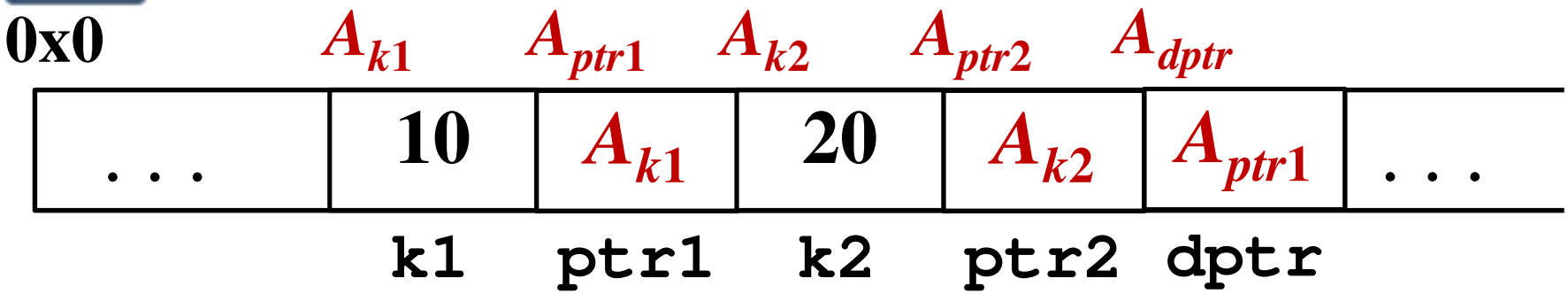
Количество знаков "*", используемых при определении переменной-указателя, определяет количество знаков "*", которое требуется для доступа к конечной ячейке, содержащей **значение**, а не **адрес**.

```
int j, *ptr=&j, **dptr=&ptr, ***tptr=&dptr;
```

```
*ptr ~ j, *dptr ~ ptr, **dptr ~ j
*tptr ~ dptr, **tptr ~ ptr, ***tptr ~ j
```



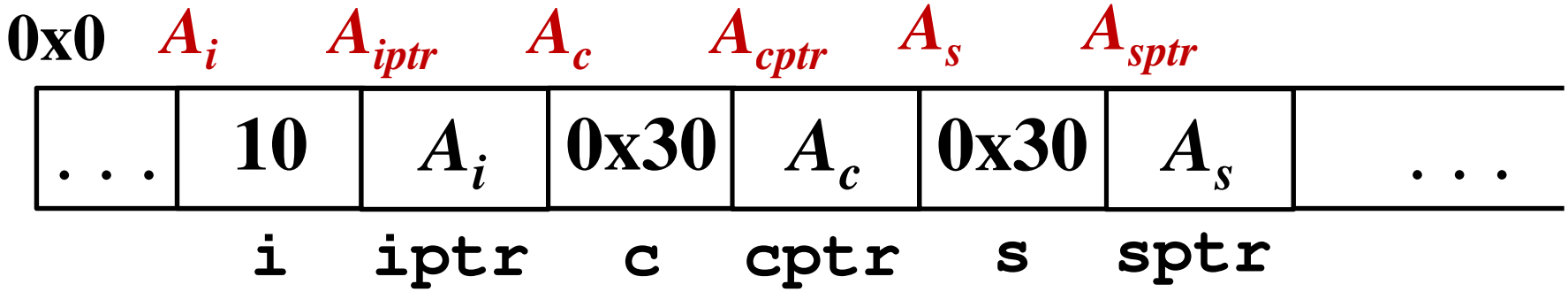
Двойной указатель (пример)




```
int k1 = 10, k2 = 20;
int *ptr1 = &k1, *ptr2 = &k2;
int **dptr = &ptr1;
**dptr = 80; // в какую ячейку запись?
dptr = &ptr2; // в какую ячейку запись?
**dptr = 100; // в какую ячейку запись?
*dptr = ptr1; // в какую ячейку запись?
**dptr = 120; // в какую ячейку запись?
```

Для чего задается тип указателя?

```
int i = 10, *iptr = &i;
char c = '0', *cptr = &c;
short s = 10, *sptr = &s;
```

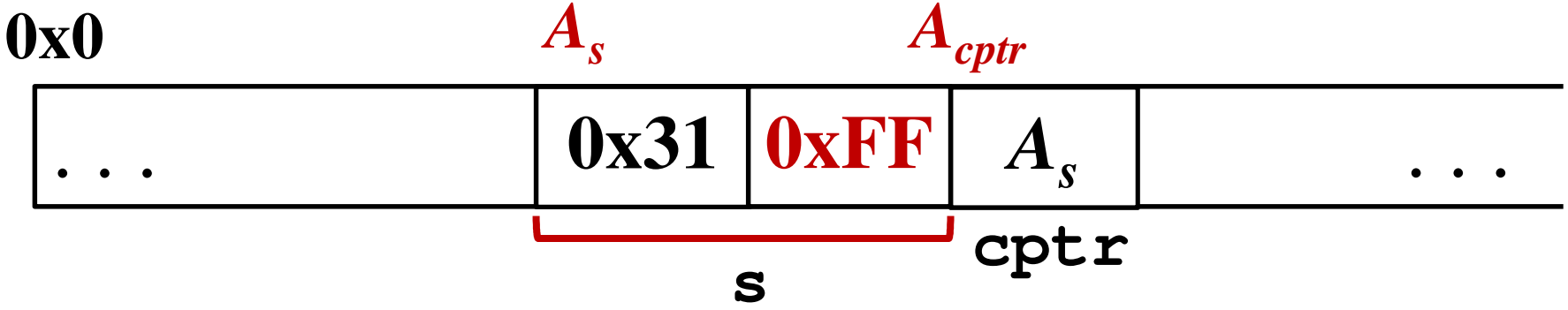


Адрес – это порядковый номер байта => **Адрес одинаков для ячеек любого размера!**

1. Копирование и разрядность данных

Для выполнения операции присваивания требуется информация о количестве байт, которые должны быть перезаписаны

```
short s = 0xFFCC;
char *cptr = (char*)&s;
*cptr = '1'; // изменен только 1-й байт!
```



Допустимые операции

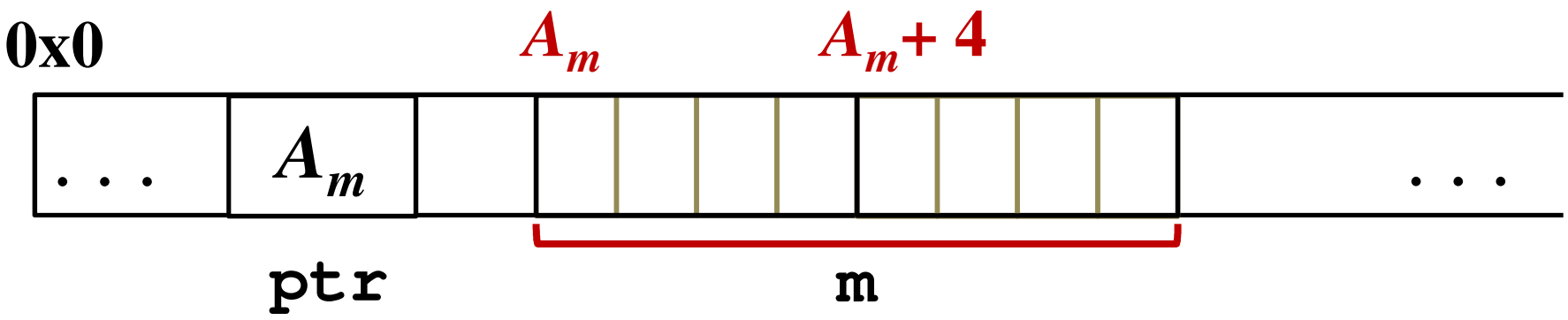
```
int *ptr1, *ptr2, j, **dptr;
```

Допустимые операции с указателями:

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

Адресная арифметика

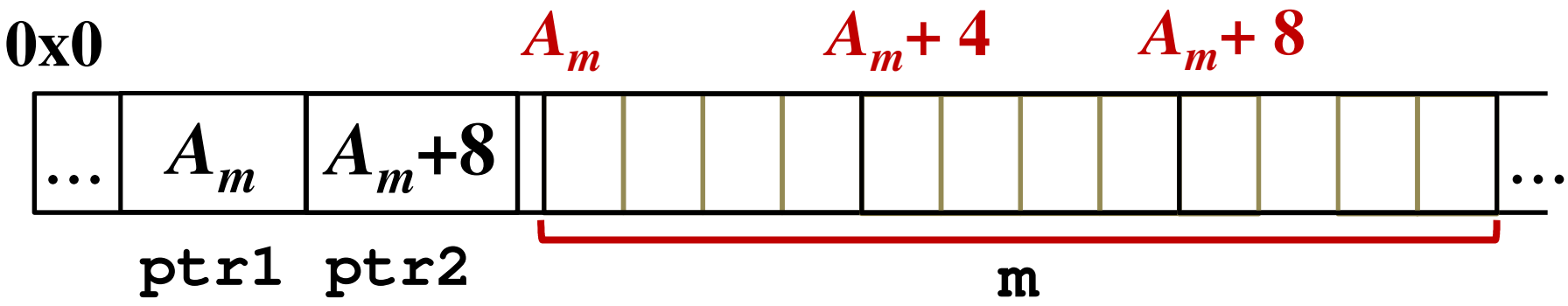
2. Адресная арифметика (сложение с целым)




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

Имя массива – УКАЗАТЕЛЬ-КОНСТАНТА на его первый элемент

2. Адресная арифметика (разность указателей)




```
int m[3], *ptr1 = &m[0], *ptr2 = &m[2];
int i = ptr2 – ptr3; // i == 2
```

Связь массивов и указателей

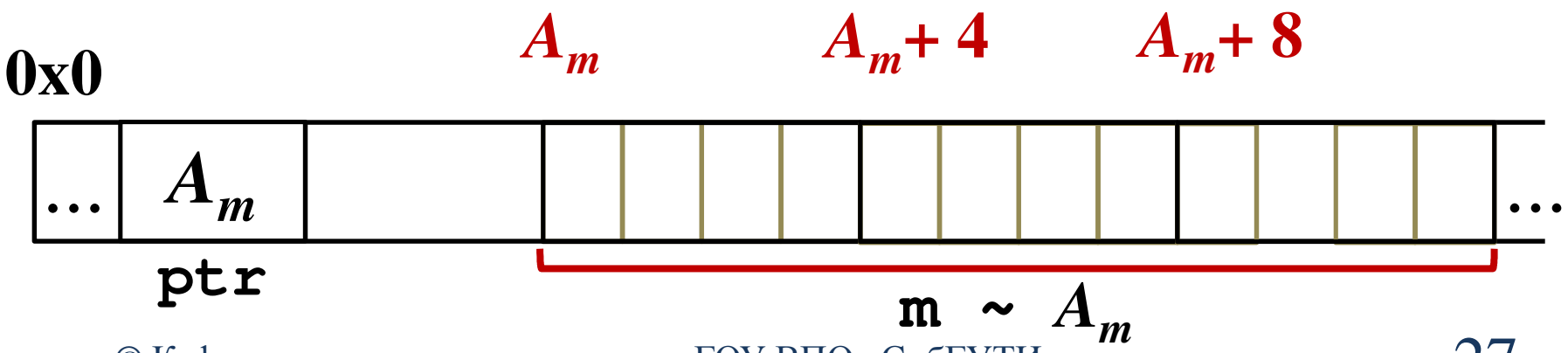
Имя массива является указателем-константой на его первый элемент.

К *N*-мерному указателю может применяться *N* операций индексации.

Например:

```
int m[3], *ptr = m, i = 2;
```

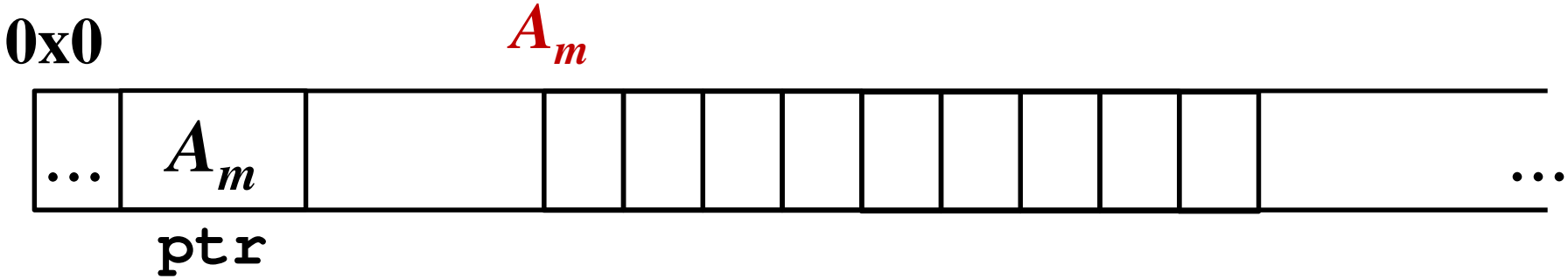
```
m[i] ~ ptr[i] ~ *(ptr + i)
```



Указатель на многомерный массив

Пусть дан массив:

```
int m[3][3];
```



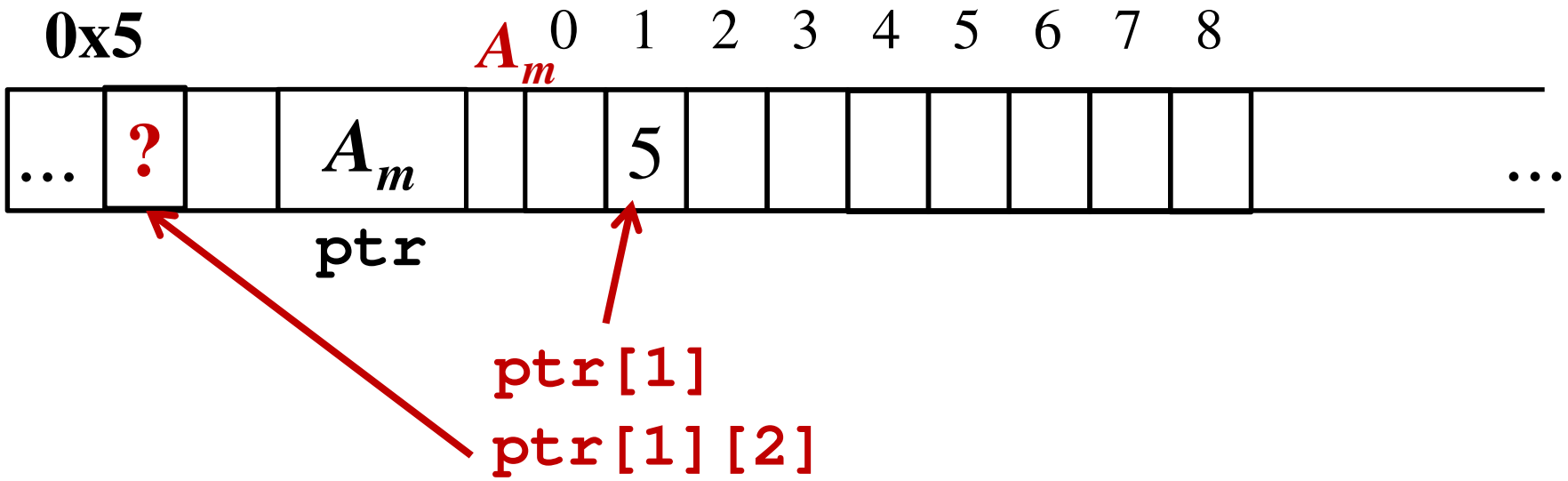

```
int **ptr1 = m, (*ptr2)[3] = m;
```

Особенность многомерного массива – его расположение в одномерной памяти программы

Это приводит к тому, что при объявлении указателя на данный массив требуется сообщить компилятору обо всех размерностях, кроме старшей

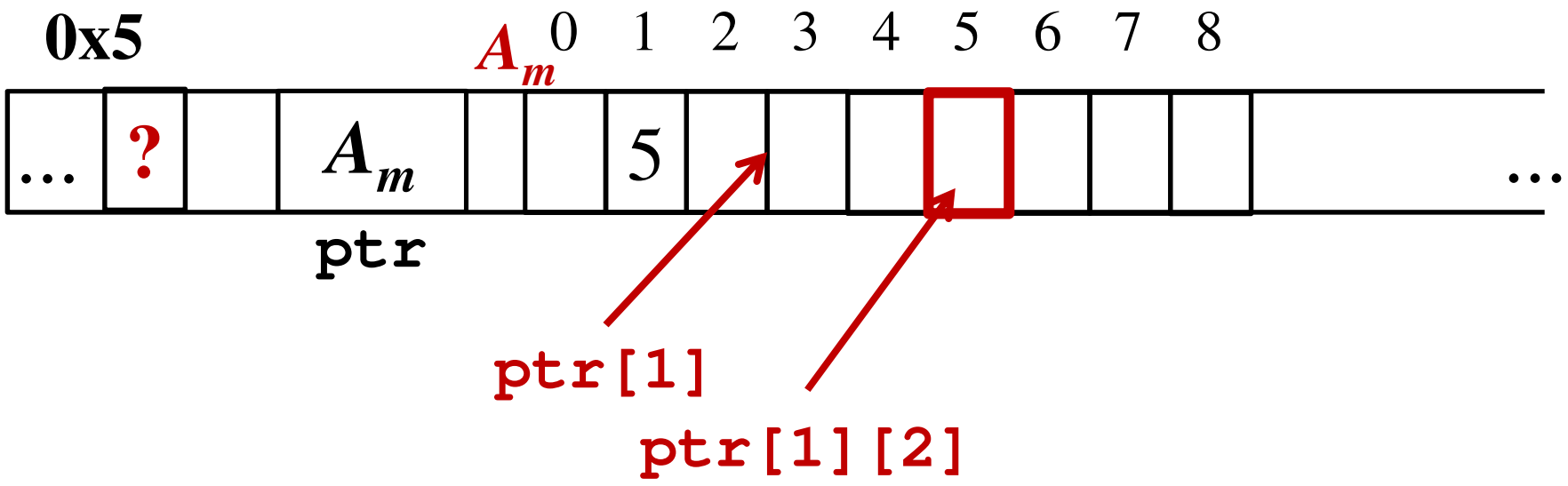
Применение двух операций индексации (неверный вариант)

```
int m[3][3], **ptr = m;
        m[i] ~ ptr[i] ~ *(ptr + i)
    ptr[i][j] ~ *(*(ptr + i) + j)
            ptr[1][2] – ?
```



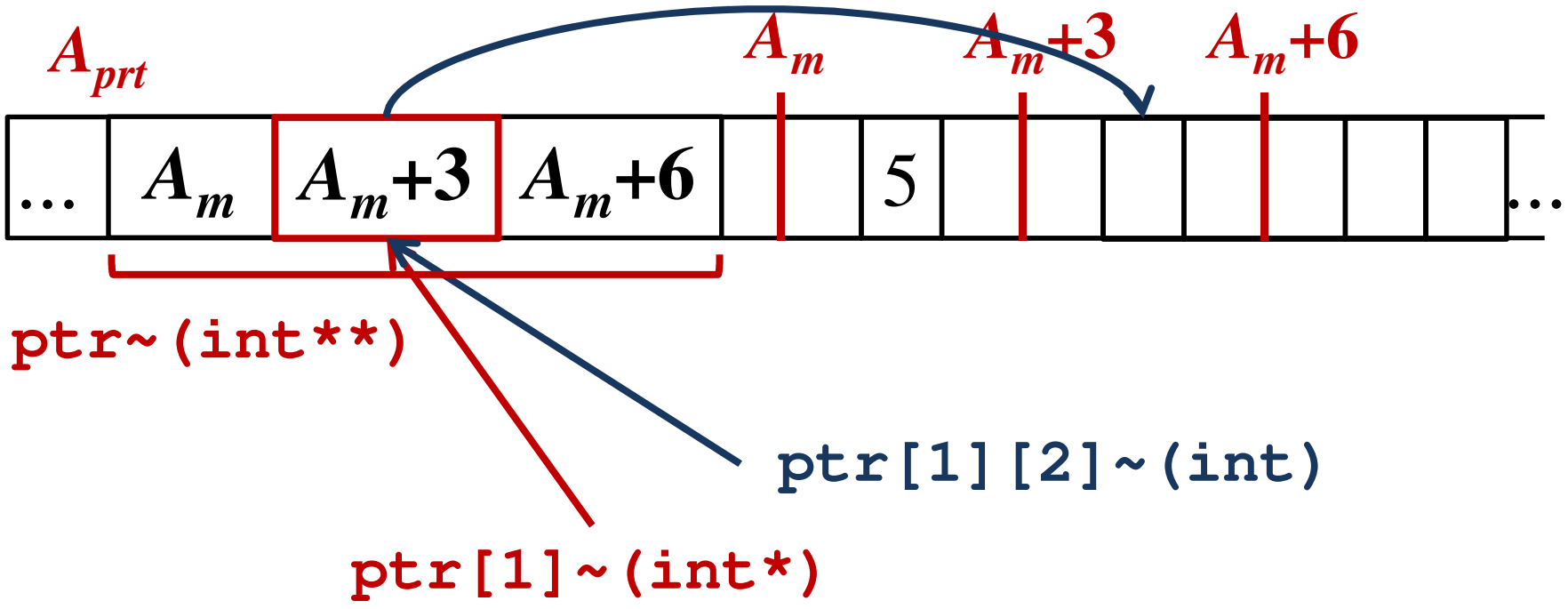
Применение двух операций индексации (правильный вариант)

```
int m[3][3], (*ptr)[3] = m;
      m[i] ~ ptr[i] ~ (ptr + i*3)
   ptr[i][j] ~ *((ptr + i*3) + j)
            ptr[1][2] – ?
```



Альтернативный способ организации двумерного массива

```
int m[3][3], *ptr[3] = {m[0],m[1],m[2]};
ptr[i] == m[i] ~ *(ptr + i)
ptr[i][j] ~ *(*(ptr + i) + j)
```



СПАСИБО ЗА ВНИМАНИЕ!