



Дисциплины
"ЯЗЫКИ ПРОГРАММИРОВАНИЯ"
"ПРОГРАММИРОВАНИЕ"

Модульное программирование

Преподаватель:

Перышкова Евгения Николаевна



Подпрограмма

- Уже в первом программируемом компьютере (аналитической машине Ч. Бэббиджа) была заложена возможность повторного использования перфокарт с наборами команд.
- В современных языках программирования набор команд, который предполагается использовать многократно, записывается в виде *подпрограммы*.
- Повторное использование отлаженного кода позволяет сократить время разработки программы, а также ее размер.
- Детали вычислений, производимых подпрограммой, заменяются в основной программе *оператором вызова* соответствующей подпрограммы.
- Такой подход улучшает читабельность программы и позволяет абстрагироваться от деталей вычислений.



Пример программы

```
#include <stdio.h>
int main()
{
    double e, x, x1, x2, eps = 1e-6, a, b, c, D, sD;
    printf("Input a, b, c: "); scanf("%lf %lf %lf", &a, &b, &c);
    D = b*b - 4*a*c;
    if( D >= 0 ){
        x1 = D;
        if ( x1 > 0 ){
            do{
                x2 = x1;
                x1 = (1.0/2)*(x1+D/x1);
            }while( (x2 - x1) >= eps );
        }
        sD = x1; x1 = (-b - sD)/(2*a); x2 = (-b + sD)/(2*a);
        if( D > 0 ) { printf("x1 = %lf, x2 = %lf\n",x1,x2); }
        else{ printf("x1 = %lf\n",x1); }
    }else{
        printf("No roots!\n");
    }
    return 0;
}
```



Пример программы (2)

```
#include <stdio.h>
int main()
{
    double e, x, x1, x2, eps = 1e-6, a, b, c, D, sD;
    printf("Input a, b, c: "); scanf("%lf %lf %lf", &a, &b, &c);
    D = b*b - 4*a*c;
    if( D >= 0 ){
        x1 = D;
        if ( x1 > 0 ){
            do{
                x2 = x1;
                x1 = (1.0/2)*(x1+D/x1);
            }while( (x2 - x1) >= eps );
        }
        sD = x1; x1 = (-b - sD)/(2*a); x2 = (-b + sD)/(2*a);
        if( D > 0 ) { printf("x1 = %lf, x2 = %lf\n",x1,x2); }
        else{ printf("x1 = %lf\n",x1); }
    }else{
        printf("No roots!\n");
    }
    return 0;
}
```

Формула Герона

$$\begin{cases} x_0 = a \\ x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right) \end{cases}$$



Пример программы (3) недостатки

```
#include <stdio.h>
int main()
{
    double e, x1, x2, eps = 1e-6, a, b, c, D, sD;
    printf("Input a, b, c: "); scanf("%lf %lf %lf", &a, &b, &c);
    D = b*b - 4*a*c;
    if( D >= 0 ){
        x1 = D;
        if ( x1 > 0 ){
            do{
                x2 = x1;
                x1 = (1.0/2)*(x1+D/x1);
            }while( (x2 - x1) >= eps );
        }
        sD = x1; x1 = (-b - sD)/(2*a); x2 = (-b + sD)/(2*a);
        if( D > 0 ) { printf("x1 = %lf, x2 = %lf\n",x1,x2); }
        else{ printf("x1 = %lf\n",x1); }
    }else{
        printf("No roots!\n");
    }
    return 0;
}
```

1. Переменные с именами x1 и x2 используются для двух разных задач.
2. То, что выделенный фрагмент кода вычисляет корень не очевидно.



Подпрограмма вычисления корня по формуле Герона

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

```
double mysqrt(double D)
```

```
{
```

```
    double eps = 1e-6;
```

```
    double x1 = D, x2;
```

```
    if ( x1 > 0 ){
```

```
        do{
```

```
            x2 = x1;
```

```
            x1 = (1.0/2)*(x1+D/x1);
```

```
        }while( (x2 - x1) >= eps );
```

```
    }
```

```
    return x1;
```

```
}
```

```
int main()
```

```
{
```

```
    double e, x1, x2;
```

```
    double a, b, c, D, sD;
```

```
    printf("Input a, b, c: ");
```

```
    scanf("%lf %lf %lf", &a, &b, &c);
```

```
    D = b*b - 4*a*c;
```

```
    if( D >= 0 ){
```

```
        sD = mysqrt(D);
```

```
        x1 = (-b - sD)/(2*a);
```

```
        x2 = (-b + sD)/(2*a);
```

```
        if( D > 0 ){
```

```
            printf("x1 = %lf, x2 = %lf\n",  
                  x1,x2);
```

```
        }else{
```

```
            printf("x1 = %lf\n",x1);
```

```
        }
```

```
        . . .
```



Свойства подпрограмм

- Каждая подпрограмма имеет один вход. При вызове подпрограммы управление передается ее *первой* инструкции.

Точка входа в
подпрограмму

```
double mysqrt(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 > 0 ) {
        do {
            x2 = x1;
            x1 = (1.0/2) * (x1 + D/x1);
        } while ( (x2 - x1) >= eps );
    }
    return x1;
}
```

Точка завершения
подпрограммы

Оператор return предполагает немедленное завершение программы и возврат ее аргумента в качестве результата



Свойства подпрограмм (2)

- Каждая подпрограмма имеет один вход. При вызове подпрограммы управление передается ее *первой* инструкции.

```
double mysqrt1(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 <= 0 )
        return x1;
    do{
        x2 = x1;
        x1 = (1.0/2)*(x1+D/x1);
    }while( (x2 - x1) >= eps );
    return x1;
}
```

```
double mysqrt(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 > 0 ){
        do{
            x2 = x1;
            x1 = (1.0/2)*(x1+D/x1);
        }while( (x2 - x1) >= eps );
    }
    return x1;
}
```




Свойства подпрограмм (3)

- На время выполнения вызываемой подпрограммы выполнение вызывающего ее модуля откладывается. В любой момент времени выполняется только одна подпрограмма.



```
...  
D = b*b - 4*a*c;  
if( D >= 0 ){  
    sD = mysqrt(D);  
    x1 = (-b - sD)/(2*a);  
    x2 = (-b + sD)/(2*a);  
    if( D > 0 ){  
        printf(...);  
    }else{  
        printf(...);  
    }  
    . . .
```

```
double mysqrt(double D)  
{  
    double eps = 1e-6;  
    double x1 = D, x2;  
    if ( x1 > 0 ){  
        do{  
            x2 = x1;  
            x1 = (1.0/2)*(x1+D/x1);  
        }while( (x2 - x1) >= eps );  
    }  
    return x1;  
}
```



Свойства подпрограмм (4)

- На время выполнения вызываемой подпрограммы выполнение вызывающего ее модуля откладывается. В любой момент времени выполняется только одна подпрограмма.



```
...  
D = b*b - 4*a*c;  
if( D >= 0 ){  
    sD = mysqrt(D);  
    x1 = (-b - sD)/(2*a);  
    x2 = (-b + sD)/(2*a);  
    if( D > 0 ){  
        printf(...);  
    }else{  
        printf(...);  
    }  
    . . .
```

```
double mysqrt(double D)  
{  
    double eps = 1e-6;  
    double x1 = D, x2;  
    if ( x1 > 0 ){  
        do{  
            x2 = x1;  
            x1 = (1.0/2)*(x1+D/x1);  
        }while( (x2 - x1) >= eps );  
    }  
    return x1;  
}
```



Свойства подпрограмм (4)

- На время выполнения вызываемой подпрограммы выполнение вызывающего ее модуля откладывается. В любой момент времени выполняется только одна подпрограмма.



```
...  
D = b*b - 4*a*c;  
if( D >= 0 ){  
    sD = mysqrt(D);  
    x1 = (-b - sD) / (2*a);  
    x2 = (-b + sD) / (2*a);  
    if( D > 0 ){  
        printf(...);  
    }else{  
        printf(...);  
    }  
    . . .
```

```
double mysqrt(double D)  
{  
    double eps = 1e-6;  
    double x1 = D, x2;  
    if ( x1 > 0 ){  
        do{  
            x2 = x1;  
            x1 = (1.0/2)*(x1+D/x1);  
        }while( (x2 - x1) >= eps );  
    }  
    return x1;  
}
```



Свойства подпрограмм (5)

- После завершения подпрограммы управление всегда возвращается в вызывающий модуль на инструкцию, следующую непосредственно за вызовом подпрограммы.

```
...  
D = b*b - 4*a*c;  
if( D >= 0 ){
```

```
    sD = mysqrt(D);
```

```
    x1 = (-b - sD)/(2*a);
```

```
    x2 = (-b + sD)/(2*a);
```

```
    if( D > 0 ){
```

```
        printf(...);
```

```
    }else{
```

```
        printf(...);
```

```
    }
```

```
    . . .
```

```
double mysqrt(double D)  
{
```

```
    double eps = 1e-6;
```

```
    double x1 = D, x2;
```

```
    if ( x1 > 0 ){
```

```
        do{
```

```
            x2 = x1;
```

```
            x1 = (1.0/2)*(x1+D/x1);
```

```
        }while( (x2 - x1) >= eps );
```

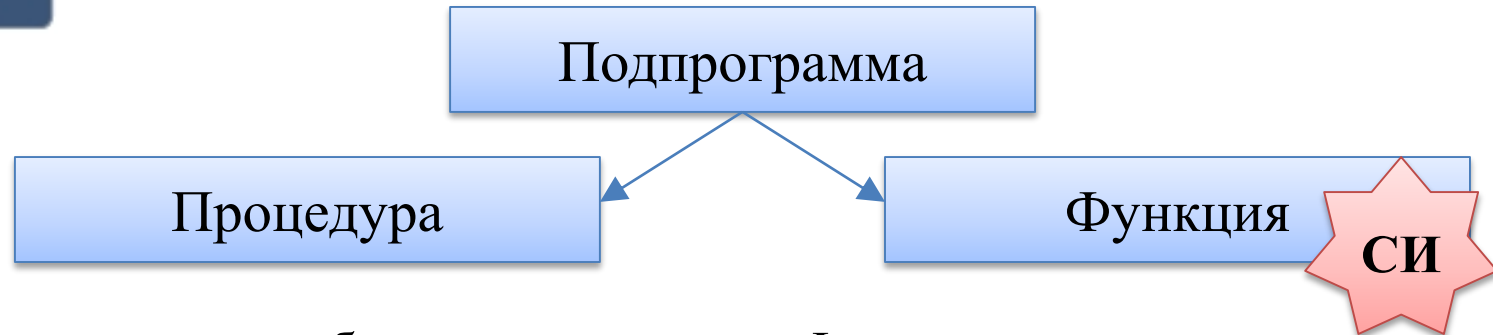
```
    }
```

```
    return x1;
```

```
}
```



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!*



Пример процедур

1. Сортировка массива m данных

Заголовок процедуры:	Вызов процедуры:
<i>Pascal:</i> procedure sort(m: array of integer);	sort(mas); sort(m); sort(array);
<i>C:</i> void sort(int m[]);	

2. Построение массива mas с разрядами числа x

Заголовок процедуры:	Вызов процедуры:
<i>Pascal:</i> procedure digits(mas: array of integer , x: integer);	digits(array, y); digits(m, x); digits(mas, y);
<i>C:</i> void digits(int mas[], int x);	



Пример процедур (2)

В обоих примерах подпрограмма изменяет данные, которые передаются в качестве аргументов. Попытка реализации аналогичной операции с помощью функций привела бы к необходимости создания копии массива, который может быть очень большим, только для того, чтобы выполнить возврат результата и поместить его обратно в исходный массив!

Заголовок процедуры:	Вызов процедуры:
<i>Pascal:</i> procedure sort(m: array of integer);	sort(mas); sort(m); sort(array);
<i>C:</i> void sort(int m[]);	

В языке СИ есть только функции. Для того, чтобы реализовать процедуру необходимо указать специальный тип данного `void`, означающий, что функция ничего не возвращает



Подпрограммы (основные определения)

С подпрограммой связано 3 понятия:

- *определение подпрограммы (subprogram definition)* – содержит информацию необходимую для вызова функции (интерфейс), а также детализацию действий, которые выполняются данной подпрограммой.
- *вызов подпрограммы (subprogram call)* – явное требование выполнить подпрограмму.
- *заголовок подпрограммы (subprogram prototype/header)* – служит признаком того, что далее последует детализация действий, реализуемых подпрограммой, а также он определяет имя функции, список ее формальных параметров и тип возвращаемого значения.



Подпрограммы (основные определения) (2)

- *профиль параметров (parameter profile)* – это количество, порядок и типы формальных параметров подпрограммы:

```
int subroutine(int x, short t, float m, double z);
```

- *протокол подпрограммы (subprogram protocol)* – это профиль ее параметров и, если это функция, тип возвращаемого ею значения.

```
int subroutine(int x, short t, float m, double z);
```

- *прототип подпрограммы (subprogram prototype)* – содержит **только** информацию необходимую для вызова функции, а именно: *имя функции и список ее параметров*.

```
int subroutine(int x, short t, float m, double z);
```



Определение и вызов подпрограммы

Определение подпрограммы

```
double mysqrt(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 > 0 ){
        do{
            x2 = x1;
            x1 = (1.0/2) * (x1+D/x1);
        }while( (x2 - x1) >= eps );
    }
    return x1;
}
```

- *вызов подпрограммы (subprogram call)* – явное требование выполнить подпрограмму.

- *определение подпрограммы (subprogram definition)* – содержит информацию необходимую для вызова функции (интерфейс), а также детализацию действий, которые выполняются данной подпрограммой.

Вызов подпрограммы

```
...
D = b*b - 4*a*c;
if( D >= 0 ){
    sD = mysqrt(D);
    x1 = (-b - sD) / (2*a);
    x2 = (-b + sD) / (2*a);
    . . .
}
```



Прототип подпрограммы

main.c

```
double mysqrt(double D);

int main()
{
    double e, x1, x2;
    double a, b, c, D, sD;
    printf("Input a, b, c: ");
    scanf("%lf %lf %lf", &a, &b, &c);
    D = b*b - 4*a*c;
    if( D >= 0 ){
        sD = mysqrt(D) ;
        x1 = (-b - sD) / (2*a);
        x2 = (-b + sD) / (2*a);
        if( D > 0 ){
            printf(...);
        }else{
            printf(...);
        }
    }

    . . .
}
```

sqrt.c

```
double mysqrt(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 > 0 ){
        do{
            x2 = x1;
            x1 = (1.0/2)*(x1+D/x1);
        }while( (x2 - x1) >= eps );
    }
    return x1;
}
```

прототип используется в ситуациях, когда вызов подпрограммы располагается до ее определения



Параметризированные вычисления

```
...  
D = b*b - 4*a*c;  
if( D >= 0 ){  
    x1 = D;  
    if ( x1 > 0 ){  
        do{  
            x2 = x1;  
            x1 = (1.0/2) * (x1+D/x1);  
        }while( (x2 - x1) >= eps );  
    }  
    sD = x1;  
    x1 = (-b - sD) / (2*a);  
    x2 = (-b + sD) / (2*a);  
    ...
```

```
...  
x3 = x1;  
if ( x3 > 0 ){  
    do{  
        x4 = x3;  
        x3 = (1.0/2) * (x3+D/x3);  
    }while( (x4 - x3) >= eps );  
}  
sx1 = x3
```

Типичные операции, такие как:

- 1) вычисление математических функций;
- 2) сортировка;
- 3) работа со строками
- 4) ввод-вывод данных

и т.д. могут использоваться многократно.

При этом копирование/размножение отложенного фрагмента кода затрудняется тем, что работка каждый раз должна производиться над разными данными.



Параметризированные вычисления (2)

```
double mysqrt(double D)
{
    double eps = 1e-6;
    double x1 = D, x2;
    if ( x1 > 0 ) {
        do {
            x2 = x1;
            x1 = (1.0/2) * (x1 + D/x1);
        } while ( (x2 - x1) >= eps );
    }
    return x1;
}
```

- Имя процедуры позволяет передать ее предназначение, что ускоряет процесс понимания чужой программы.
- В вызывающей процедуре нет необходимости отслеживать отсутствие конфликта переменных.

- Процедура позволяет реализовать набор операций однократно.
- Все переменные, необходимые только для выполнения процедуры описываются внутри и не смешиваются с другими.
- Параметры процедуры имеют одинаковые имена, поэтому код не нужно адаптировать под конкретную ситуацию.

```
...
D = b*b - 4*a*c;
if( D >= 0 ) {
    sD = mysqrt(D);
    x1 = (-b - sD) / (2*a);
    x2 = (-b + sD) / (2*a);
    ...
    sx1 = mysqrt(x1);
}
```



РБНФ определения функции

ОпределениеФункции =

ТипРезультата Имя "(" СписокФормПарам ")" " {"
{ОператорОписания} {Оператор} "}" .

ТипРезультата = ТипДанного .

Имя = Идентификатор .

СписокФормПарам =

[ТипДанного Идентиф {", " ТипДанного Идентиф }] .

ТипДанного = БазовыйТип | ПользовательскийТип .

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



РБНФ вызова функции

ВызовФункции =

[Идентиф=] Имя "(" СписокФактПарам ")" ";" .

Идентиф = Идентификатор

Имя = Идентификатор

СписокФактПарам = [Идентиф { "," Идентиф }]

ТипДанного = БазовыйТип | ПользовательскийТип

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




РБНФ прототипа функции

Прототип =

ТипРезультата Имя "(" СписокФормПарам ")" ";" .

ТипРезультата = ТипДанного .

Имя = Идентификатор .

СписокФормПарам =

[ТипДанного Идентиф {", " ТипДанного Идентиф }] .

ТипДанного = БазовыйТип | ПользовательскийТип .

```
int sum(int i,int j);
```

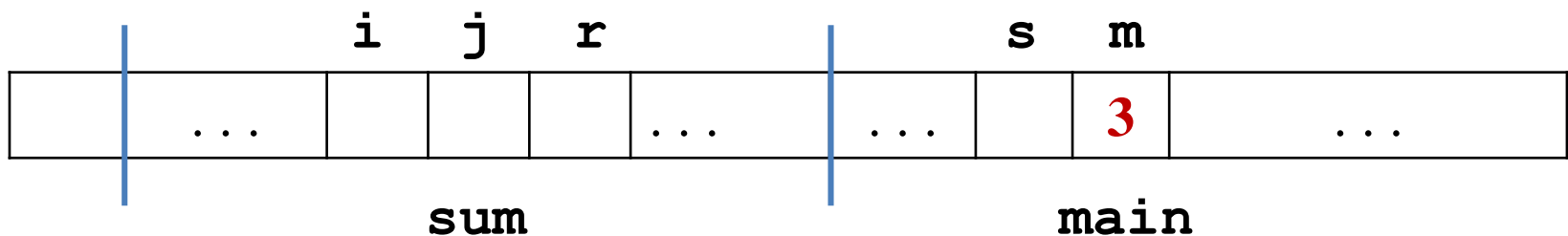


Передача параметров по значению (запуск функции main)

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);
}
```





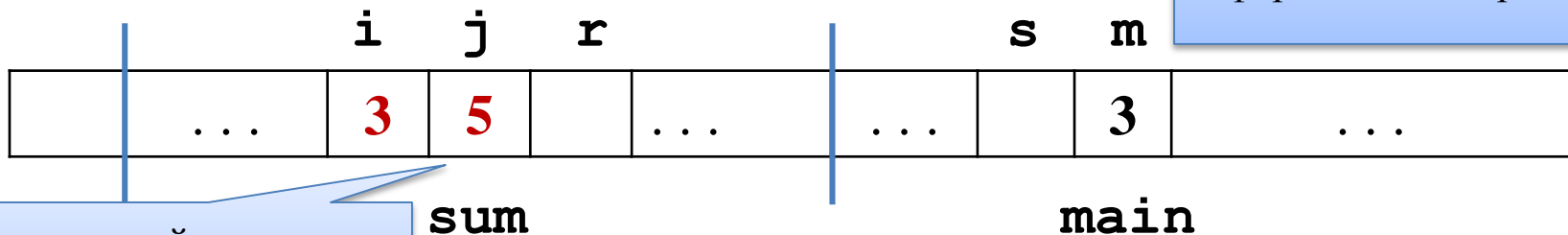
Передача параметров по значению (вызов sum)

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*

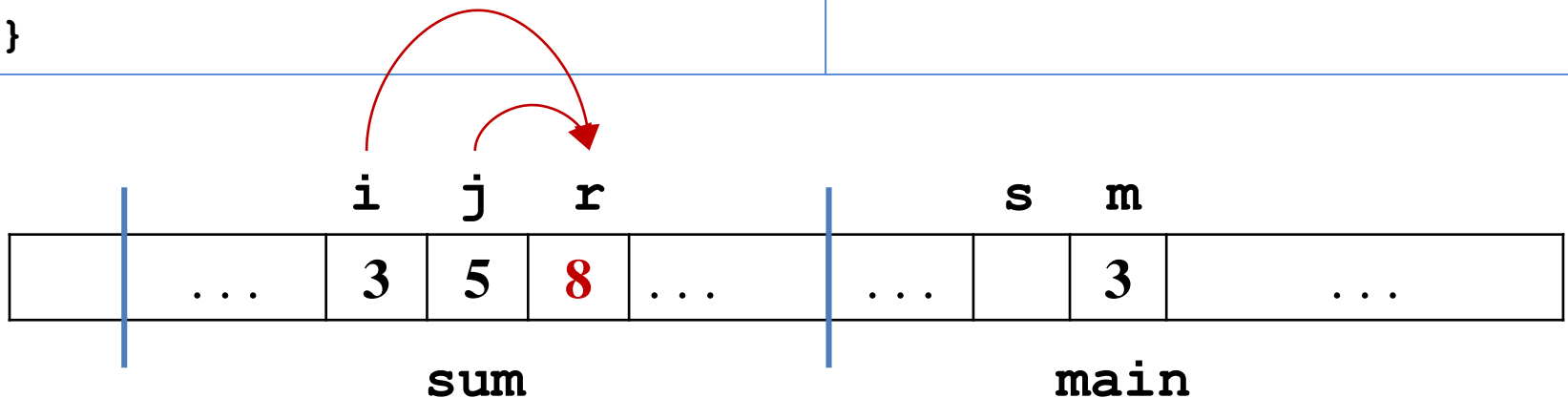


Передача параметров по значению (выполнение тела функции sum)

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;
}
```

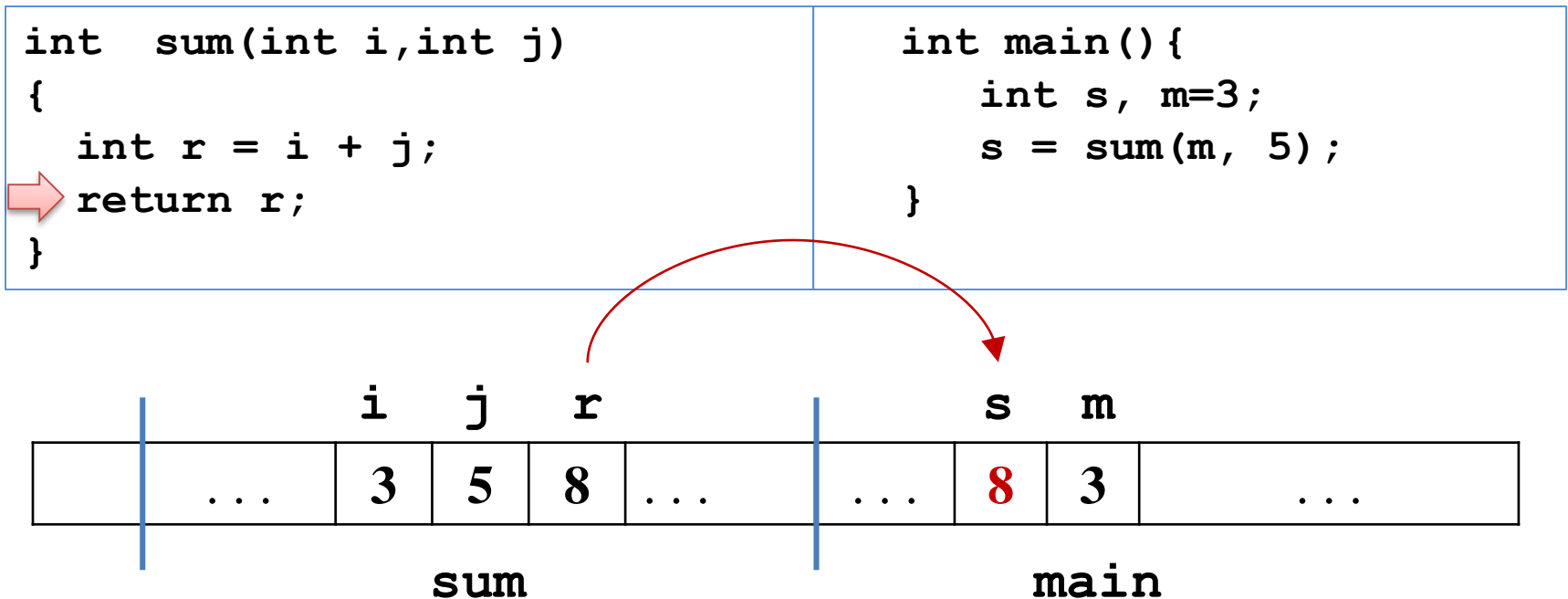
```
int main() {
    int s, m = 3;
    s = sum(m, 5);
}
```





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.



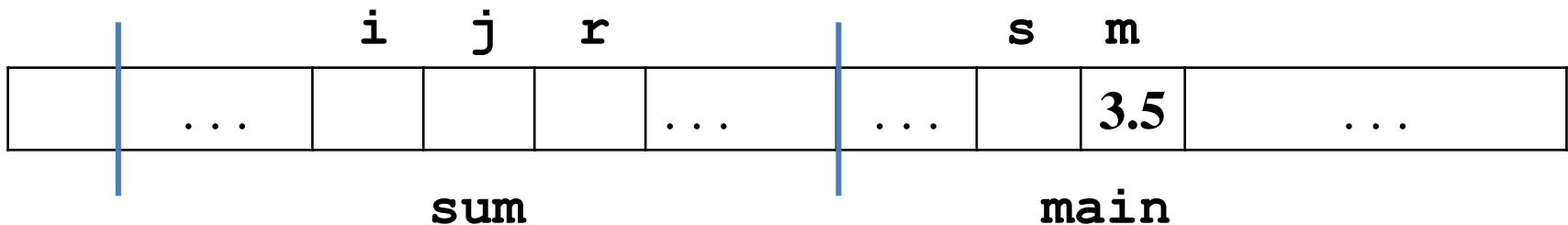


Приведение типов фактических параметров к формальным (вызов main)

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);
}
```



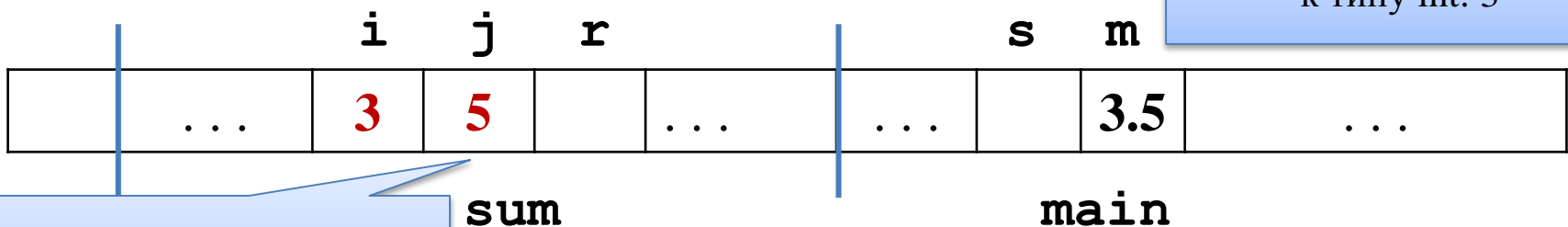


Приведение типов фактических параметров к формальным (вызов sum)

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 m = 5.5 приводится к типу int: 3

(float)5.5 приводится к типу int: 5

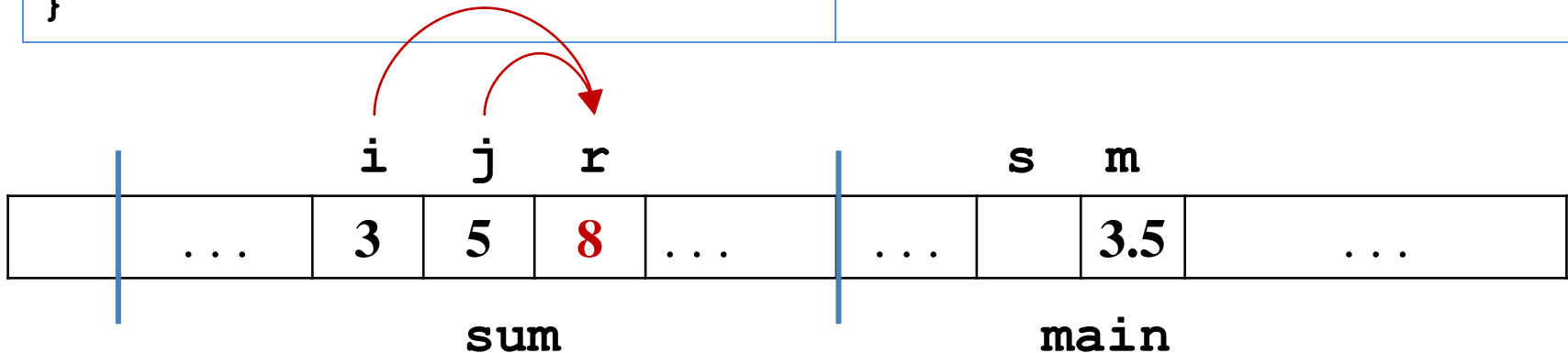


Приведение типов фактических параметров к формальным (тело sum)

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);
}
```



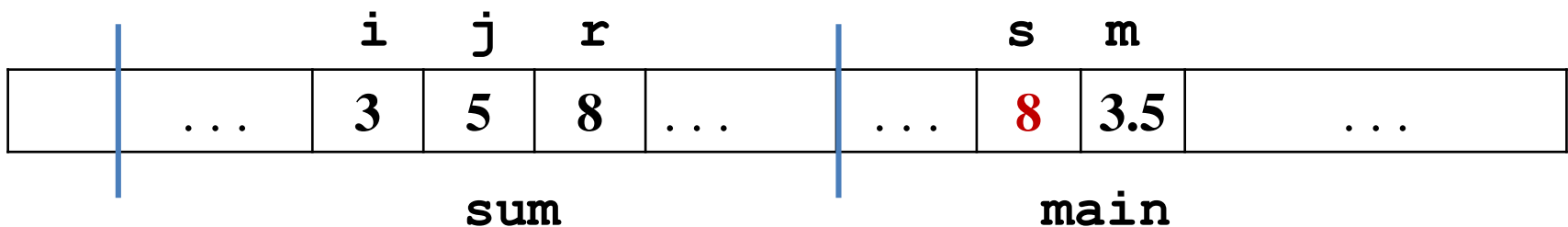


Приведение типов фактических параметров к формальным (возврат из sum)

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.

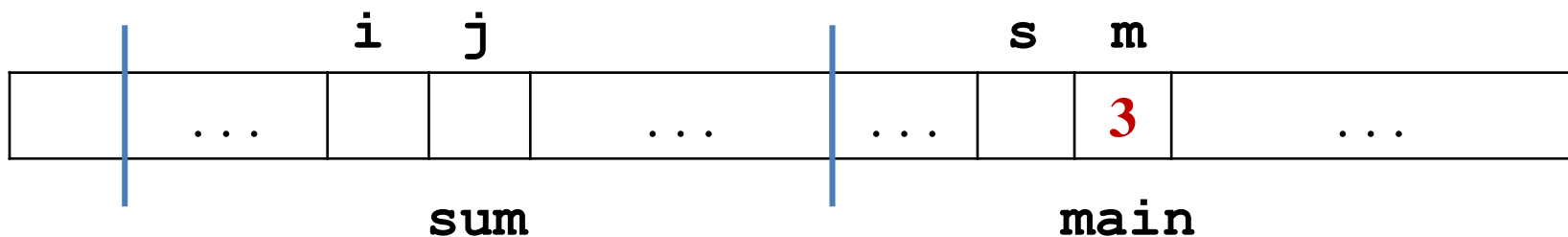


Пример **неправильной** процедуры (запуск функции main)

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);
    ...
}
```





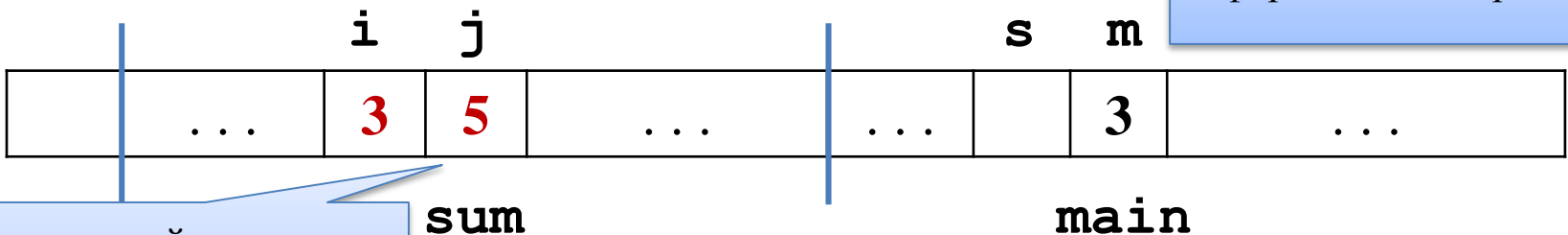
Пример **неправильной** процедуры (запуск функции main)

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);
  ...
}
```



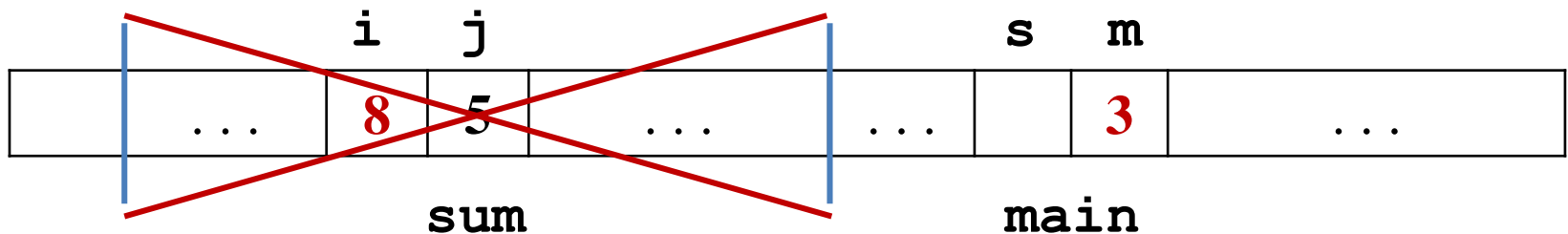


Пример **неправильной** процедуры (запуск функции main)

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);
    ...
}
```



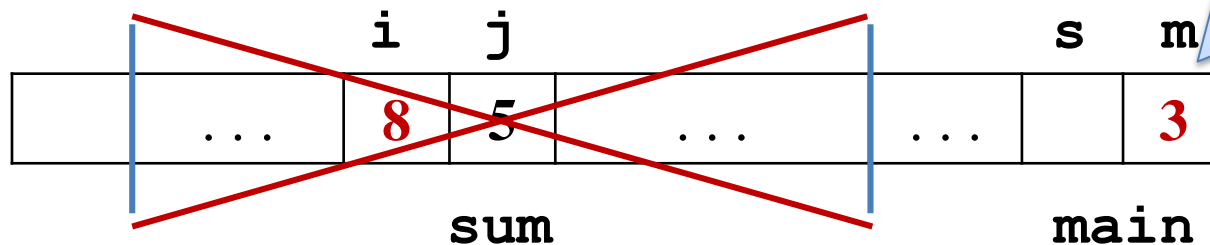


Пример **неправильной** процедуры (запуск функции main)

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!

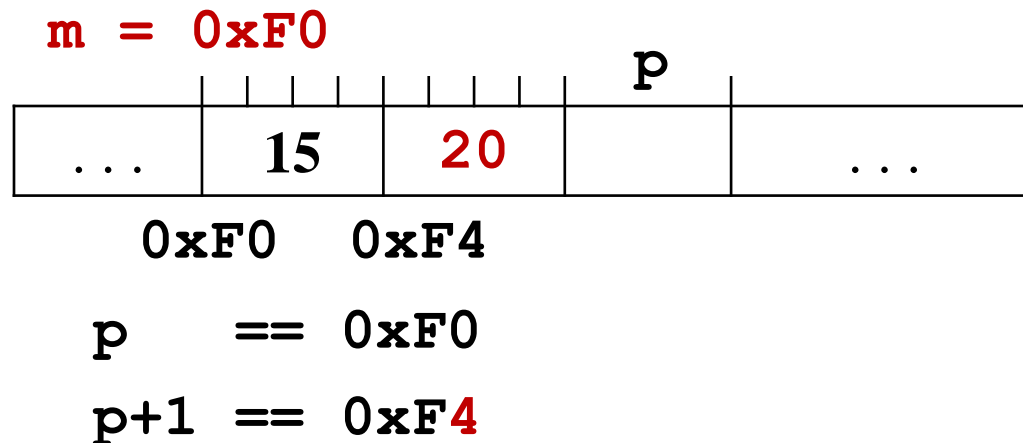
0xF1



Адресная арифметика (сложение с целым)

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

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



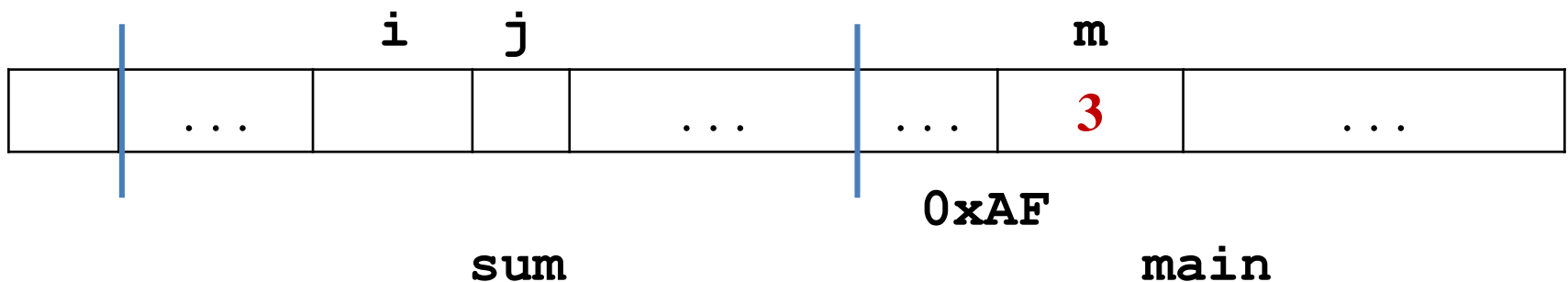


Пример **правильной** процедуры (запуск функции main)

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*

0xAF

sum

main



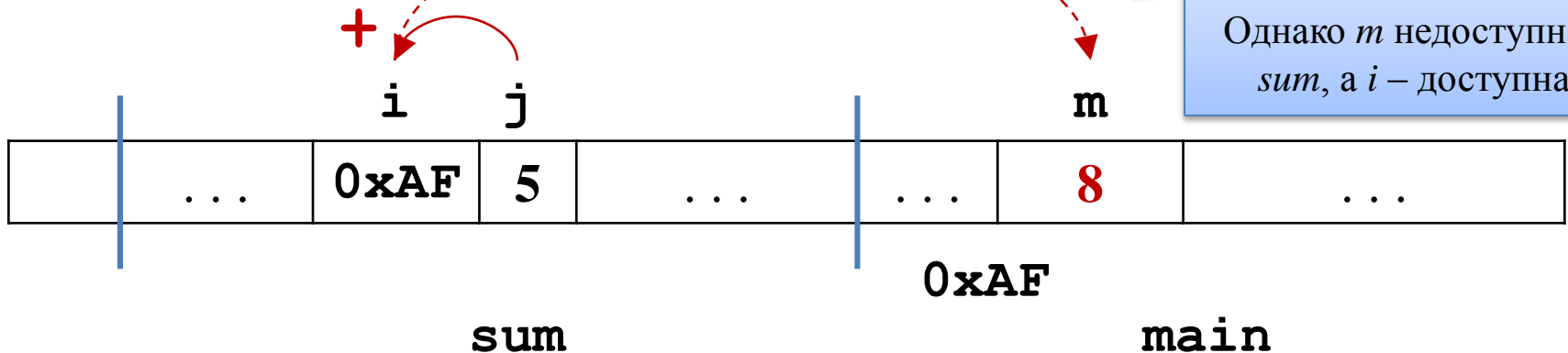
Пример **правильной** процедуры (выполнение тела `sum`)

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.





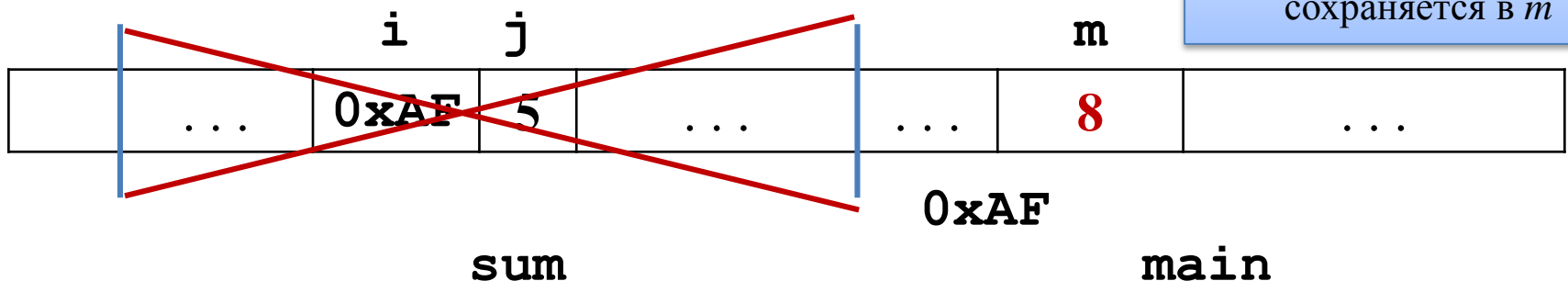
Пример **правильной** процедуры (завершение `sum`, переход в `main`)

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*



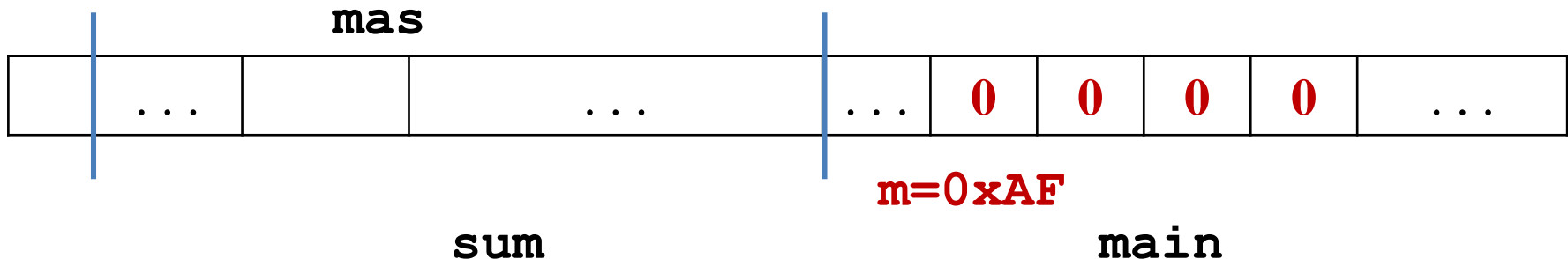


Процедура обработки массива (запуск функции main)

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);  
    ...  
}
```





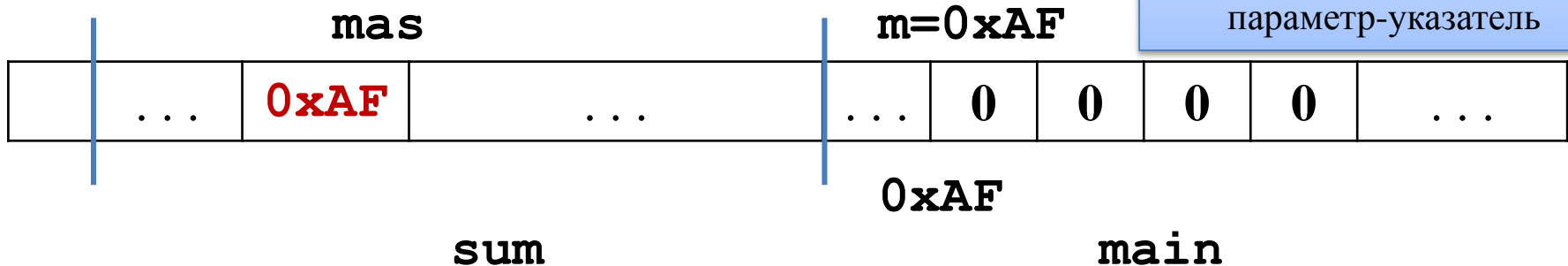
Процедура обработки массива (запуск функции main)

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





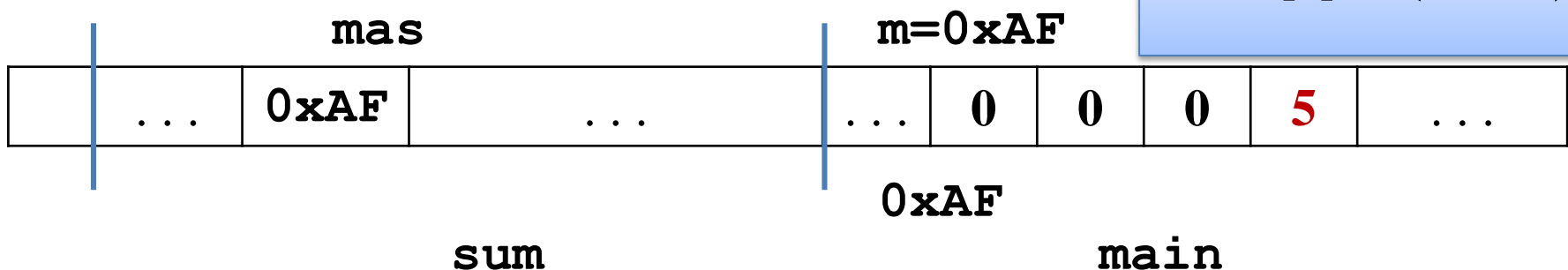
Процедура обработки массива (выполнение тела sum)

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*

