C arrays

Arrays

- Arrays are defined to be a sequence/set of data elements of the same type. Having an array, each array element can be accessed by its position in the sequence of the array.
- **Decleration** of the Arrays: Any array declaration contains: the ***array name***, the ***element type*** and the ***array size***.
- **Examples:**
- int a[20], b[3],c[7];
- float f[5], c[2];
- char m[4], n[20];

Arrays

- **Initialisation** of an array is the process of assigning initial values. Typically declaration and initialisation are combined.
- **Examples:**
- int a[4]={1,3,5,2};
- float, b[3]={2.0, 5.5, 3.14};
- char name[4]= {'E','m','r','e'};
- int c[10]={0};

Example

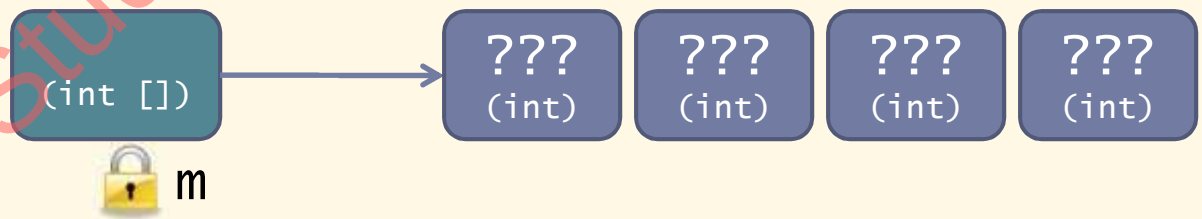
- Write a program to calculate and print the average of the following array of integers.
- (4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

```
#include<stdio.h>
void main()
{
int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0;
float av;
for(i=0,i<=size-1;i++)
sum = sum + x[i];
av = (float)sum/size;
printf("The average of the numbers=%.2f\n", av);
}
```

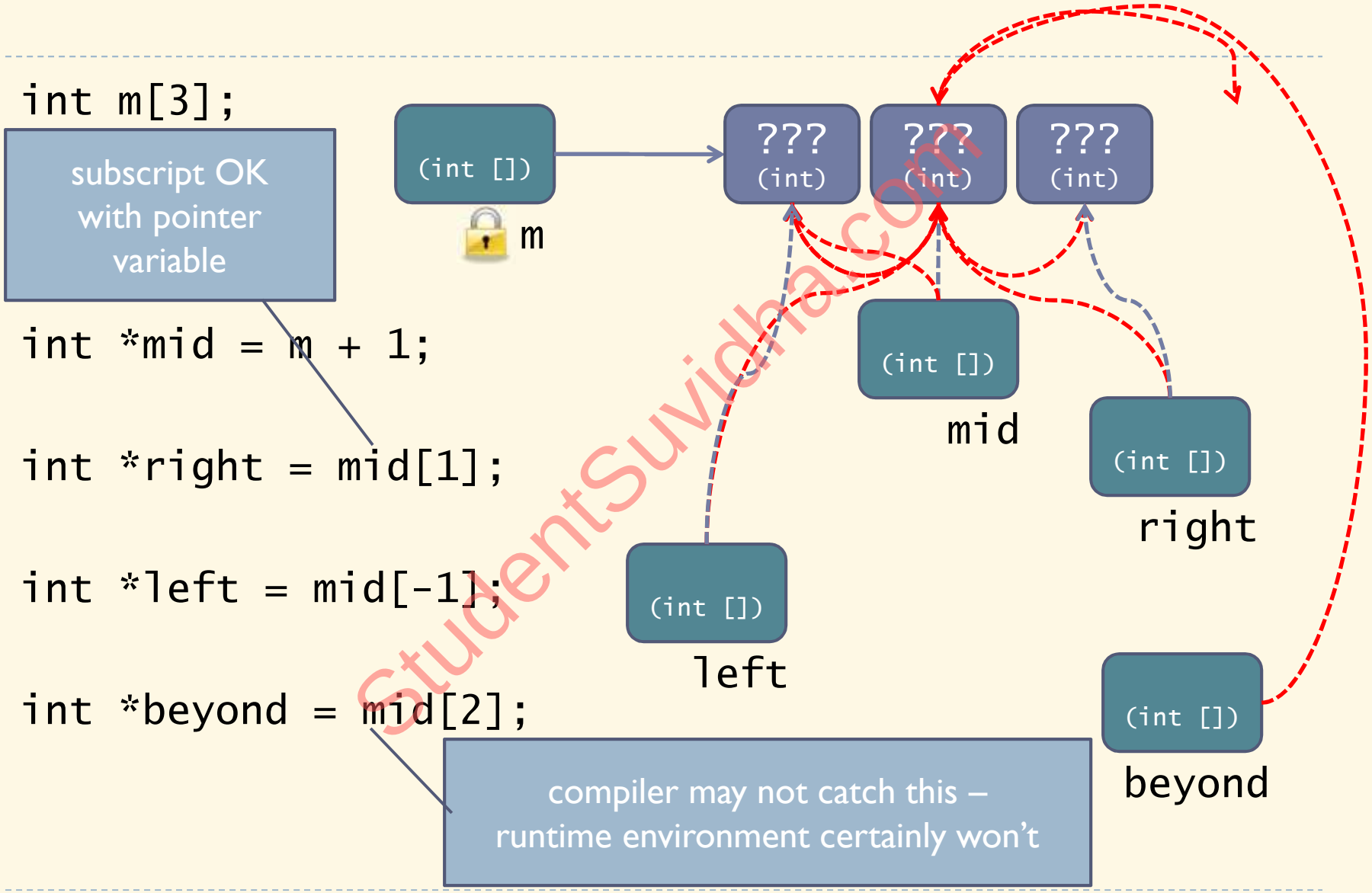
Review of arrays

- There are no array variables in C – only array *names*
  - Each name refers to a constant pointer
  - Space for array elements is allocated at declaration time
- Can't change where the array name refers to…
  - but you can change the array elements, via pointer arithmetic

```
int m[4];
```



Array names and pointer variables

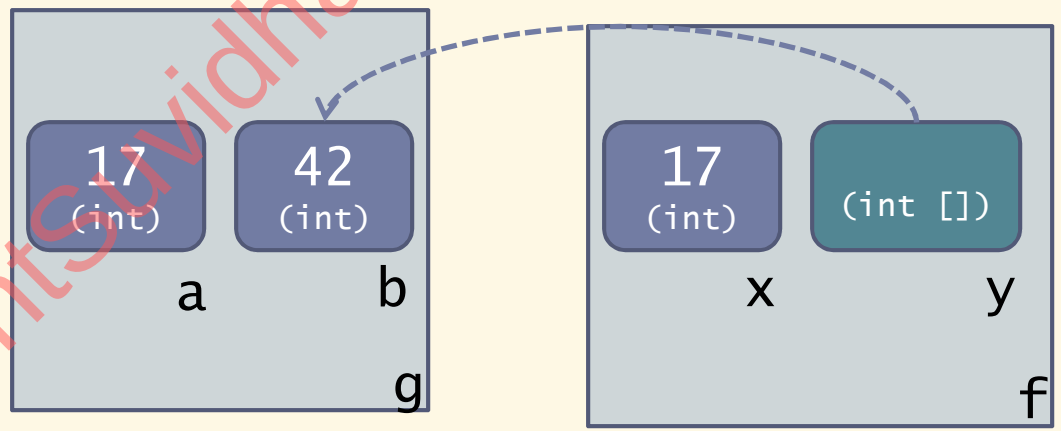


Array names as function arguments

- In C, arguments are passed "by value"
  - A temporary copy of each argument is created, solely for use within the function call

```
void f(int x, int *y) { … }

void g(…) {
  int a = 17, b = 42;
  f(a, &b);
  …
}
```



- Pass-by-value is "safe" in that the function plays only in its "sandbox" of temporary variables –
  - can't alter the values of variables in the callee (except via the return value)

Array names as function arguments

- But, functions that take arrays as arguments can exhibit *what looks like* "pass-by-reference" behavior, where the array passed in by the callee does get changed
  - Remember the special status of arrays in C –

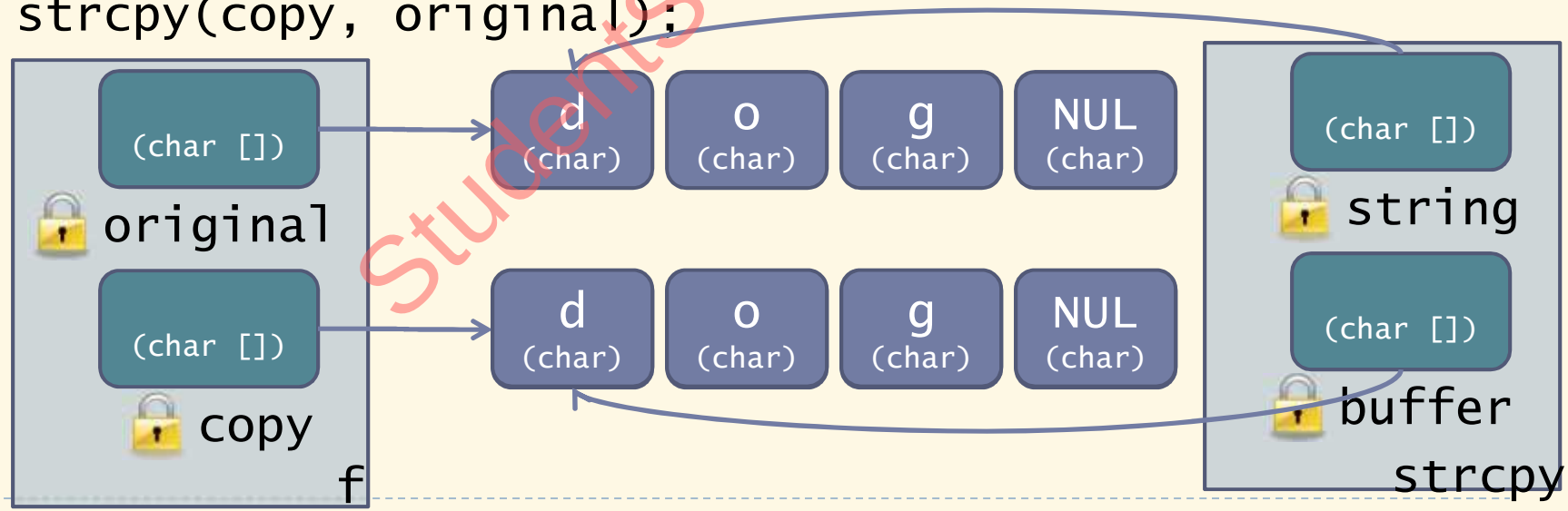
    They are basically just pointers.
  - So arrays are indeed passed by value –

    but only the pointer is copied, not the array elements!
    - Note the advantage in efficiency (avoids a lot of copying)
    - But – the pointer copy points to the same elements as the callee's array
    - These elements can easily be modified via pointer manipulation

Array names as function arguments

- The `strcpy` "string copy" function puts this "pseudo" call-by-reference behavior to good use

```
void strcpy(char *buffer, char const *string);
void f(…) {
   char original[4] = "dog";
   char copy[4];
   strcpy(copy, original);
}
```



When can array size be omitted?

- There are a couple of contexts in which an array declaration need not have a size specified:
  - Parameter declaration:

```
int strlen(char string[]);
```

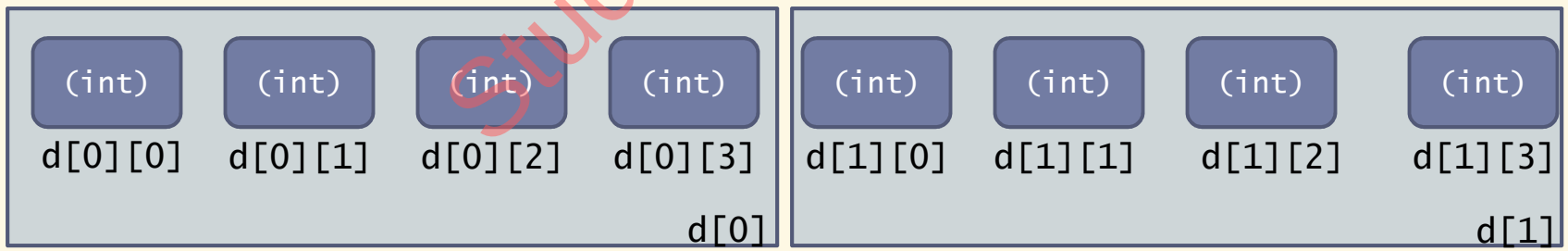
    - As we've seen, the elements of the array argument are not copied, so the function doesn't need to know how many elements there are.
  - Array initialization:

```
int vector[] = {1, 2, 3, 4, 5};
```

    - In this case, just enough space is allocated to fit all (five) elements of the initializer list

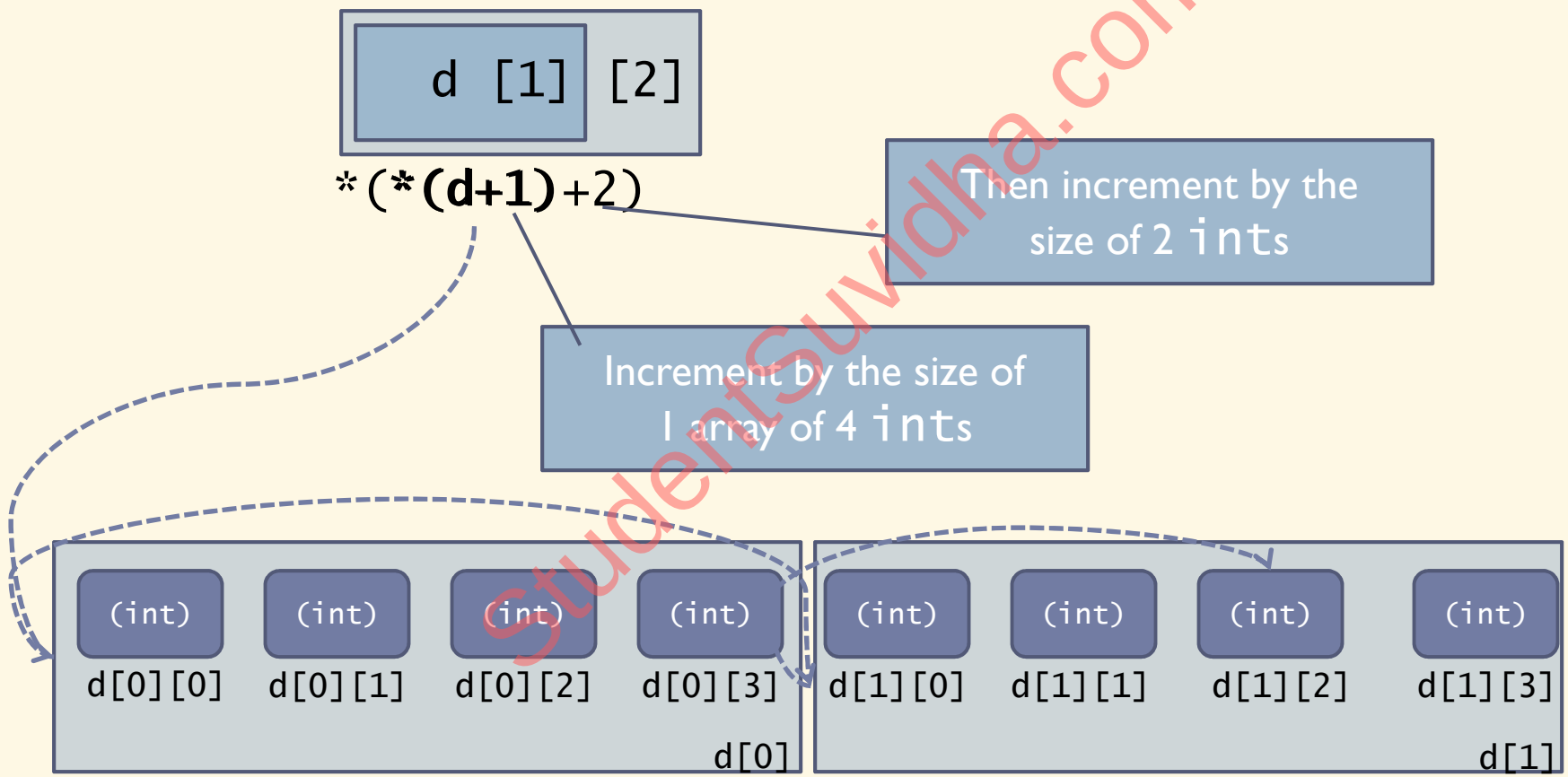
Multidimensional arrays

- How to interpret a declaration like:
  `int d[2][4];`
- This is an array with two elements:
  - Each element is an array of four `int` values
- The elements are laid out sequentially in memory, just like a one-dimensional array
  - Row-major order: the elements of the *rightmost* subscript are stored contiguously



Subscripting in a multidimensional array

```
int d[2][4];
```



Why do we care about storage order?

- If you keep within the "paradigm" of the multidimensional array, the order doesn't matter…
- But if you use tricks with pointer arithmetic,

  it matters a lot
- It also matters for initialization
  - To initialize d like this:

0	1	2	3
4	5	6	7

    - use this:

```
int d[2][4] = {0, 1, 2, 3, 4, 5, 6, 7};
```

    - rather than this

```
int d[2][4] = {0, 4, 1, 5, 2, 6, 3, 7};
```

Multidimensional arrays as parameters

- Only the first subscript may be left unspecified

```
void f(int matrix[][10]);      /* OK */
void g(int (*matrix)[10]);     /* OK */
void h(int matrix[][]);        /* not OK */
```

- Why?
  - Because the other sizes are needed for scaling when evaluating subscript expressions (see slide 10)
  - This points out an important drawback to C:

    Arrays do not carry information about their own sizes!

    If array size is needed, you must supply it somehow

    (e.g., when passing an array argument, you often have to pass an additional "array size" argument) – bummer

Assignment

- What is array? Write a program to multiply two matrices.