

MAN, I SUCK AT THIS GAME.  
CAN YOU GIVE ME  
A FEW POINTERS?

I HATE YOU.



0x3A28213A  
0x6339392C,  
0x7363682E.



# C++

Gordon Erlebacher  
++ Peter Beerli

# C++

- Typed language
  - every variable must be declared
  - every variable has a type (int, string, etc.)
  - C++ is compiled to object code (contains information about symbols and relative locations in memory)
  - Multiple object files are linked into an executable (machine code)

# Program Structure

- `int main(char* argc, char** argv) {}`
  - main function is a requirement
- `main()` calls other functions, defines, classes, etc.
- structure of C++ program is very similar to structure of a python program
- C++ is less general and less powerful than Python
- C++ is often much faster than Python (unless modules like numpy and scipy are used, which are optimized and use underlying C/C++ code)

# Second half of course

*Fortran*

- You all know Java or C++, therefore, we will concentrate on:
  - use of pointers and references
  - use of preprocessing macros
  - function and operator overloading
  - templates
  - STL library
  - wrapping C++/C functions with Python

# Anatomy of a simple C++ program

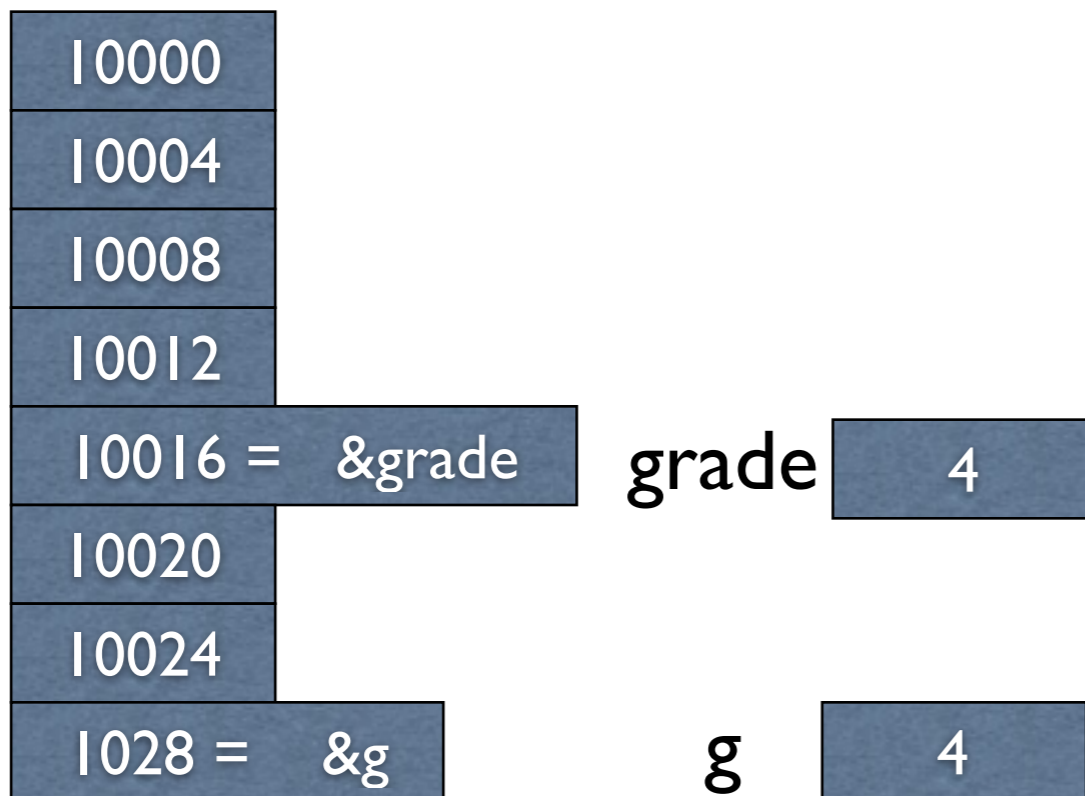
```
1 #include <iostream>
2 // example C++ code
3 // (c) Peter Beerli
4 //
5 // function square takes a double value and returns the square
6 double square(double x)
7 {
8     return x*x;
9 }
10 /*
11 The main is mandatory!
12 */
13 int main()
14 {
15     double x = 13;
16     std::cout << "Hello World!\n";
17     std::cout << "Square of " << x << " is " << square(x) << std::endl;
18     return 0;
19 }
20
```

```
arzak:examples>g++ -std=c++11 example.cpp -o ex
arzak:examples>ex
Hello World!
Square of 13 is 169
```

# Pointers

- What is a pointer?
  - a pointer is a memory address
  - `int grade; // semi-colon necessary`  
`int* grade_p; // pointer`  
`// comment`  
`/* another comment */`

# Pointer



```
int grade = 4;  
int* grade_ptr = &grade;
```

```
// grade_ptr = 10016
```

```
int g = *grade_ptr; // 4  
int* gp = &g;
```



```
int* p; // pointer to an integer
int i; // integer value
```

You turn a pointer into a value with `*`:

```
int i2 = *p; // integer value
```

You turn a value into a pointer with `&`:

```
int* p2 = &i; // pointer to an integer
```

Edit: In the case of arrays, they are treated very much like pointers. If you think of them as pointers, you'll be using `*` to get at the values inside of them as explained above, but there is also another, more common way using the `[]` operator:

```
int a[2]; // array of integers
int i = *a; // the value of the first element of a
int i2 = a[0]; // another way to get the first element
```

To get the second element:

```
int a[2]; // array
int i = *(a + 1); // the value of the second element
int i2 = a[1]; // the value of the second element
```

So the `[]` indexing operator is a special form of the `*` operator, and it works like this:

```
a[i] == *(a + i); // these two statements are the same thing
```

```

#include <stdio.h>

// or
// int main(int argc, char** argv)
int main()
{
    int grade = 4;
    int* grade_p = &grade;

    printf("grade= %d\n", grade);
    printf("&grade= %ld\n", (long) grade_p);
    printf("sizeof(grade_p)= %d\n", (int) sizeof(grade_p));

    int g = *grade_p;
    int* gp = &g;
    printf("g= %d\n", g);
    printf("&g= %ld\n", (long) &g);

    return 0;
}

```

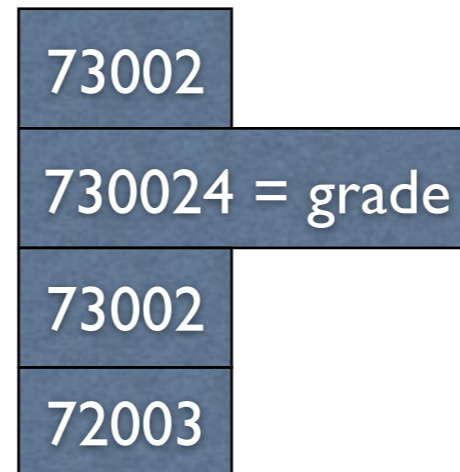
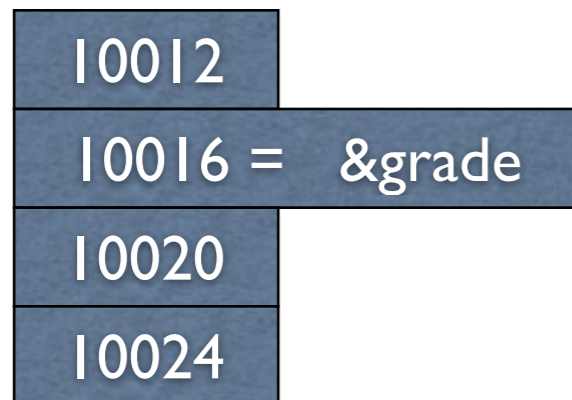
```

nagal:examples>ls
gugus.cpp  gugus.h
nagal:examples>g++ -o gugus gugus.cpp
nagal:examples>gugus
grade= 4
&grade= 140734634980712
sizeof(grade_p)= 8
g= 4
&g= 140734634980700

```

# new operator

- `float* grade = new float;`
  - allocates memory for a float
  - float takes `sizeof(float)` bytes (8 on modern macOSX)
  - the address



# Sizes

- `int grade` // uses up 4 bytes of memory  
`int* grade_p` // uses up 8 bytes of memory
- How many bytes are used by `grade`?  
Answer: 4 (`sizeof(grade)=4`)
- How many bytes are used by `&grade`?  
Answer: 8 (`sizeof(grade_p)=8`)

# Double pointer

- `float a;`
- `float* b; // pointer to b`
- `float** c; // pointer to pointer to b`

# 1D Arrays

- `float* a = new float [10];`
- `float a[10];`
- `float b[]; // illegal (unknown size)`
- `float c[] = {1.,32.,-2.5}; // ok`

# float\* a

- float\* a;  
a = new float [10];
- float\* b = new float [10];
- &a : address of a
- a : address of a[0]
- a[0] : content of a
- \*a : content of a
- \*(&a) : content of &a = content of a

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

```
int main()  
{
```

```
    float* a;  
    a = new float [10];  
    a[0] = 45.;  
    a[1] = -145.;
```

```
    float* b = new float [10];
```

```
    printf("&a = %ld\n", (long) a);  
    printf("a = %ld\n", (long) a);  
    printf("a[0] = %f, *a= %f\n", a[0], *a);  
    printf("a[1] = %f\n", a[1]);  
    printf("&a[0] = %ld\n", (long) &a[0]);  
    printf("&a[1] = %ld\n", (long) &a[1]);  
}
```

```
&a = 140473360469568  
a = 140473360469568  
a[0] = 45.000000, *a= 45.000000  
a[1] = -145.000000  
&a[0] = 140473360469568  
&a[1] = 140473360469572
```

**4 bytes per float**



# float\*\* a

- Pointer to pointer to float
- **float\*\* a** or  
float \*\*a
- Both are admissible
- The **second** is preferred by me, but some people like
  - float\*\* can be considered as a new type, but see this [!!!!]
  - float \*\*a, \*\*b; // two float\*\* variables
  - float\*\* a, b; // a is float\*\*, but b is float.
  - float\*\* is not really a type
  - Way out: typedef

# Typedef

```
#include <stdio.h>

typedef float** FLOAT_PP;

FLOAT_PP a, b; // two float**
float** c, d;  // one float** and one float

int main()
{
    a = new float* [10];
    b = new float* [10];

    c = new float* [10];
    d = new float* [10]; // ILLEGAL (d is a float)
}
```

typedef is very useful to create new types

Most useful when using the Standard Template Library

```

int main()
{
    float** a;
    a = new float* [4];
    a[0] = new float [5];
    a[1] = new float [6];

    a[0][0] = 45.;
    a[0][1] = -145.;
    a[1][0] = 32.;
    a[1][1] = 35.;

    printf("sizeof(float)= %d\n", (int) sizeof(float));
    printf("&a= %ld\n", (long) a);
    printf("a= %ld\n", (long) a);
    printf("*a= %ld\n", (long) *a);
    printf("**a= %f, a[0][0]= %f\n", (float) **a, (float) a[0][0]);
    printf("*a[0]= %f\n", (float) *a[0]);
    printf("a[0]= %ld, a[1]= %ld\n", (long) a[0], (long) a[1]);
    printf("&a[0]= %ld\n", (long) &a[0]);
    printf("&a[1]= %ld\n", (long) &a[1]);
    printf("&a[0][0]= %ld\n", (long) &a[0][0]);
    printf("&a[1][0]= %ld\n", (long) &a[1][0]);
}

```

```

./a.out
sizeof(float)= 4
&a= 4296016000
a= 4296016000
*a= 4296016032
**a= 45.000000, a[0][0]= 45.000000
*a[0]= 45.000000
a[0]= 4296016032, a[1]= 4296016064
&a[0]= 4296016000
&a[1]= 4296016008
&a[0][0]= 4296016032
&a[1][0]= 4296016064

```

# Creation of a triangular matrix

- Assume  $n$  rows
- row 1: 1 column
- row 2: 2 columns
- ...
- row  $n$ :  $n$  columns

```
float** triangular(int n)
{
    float** tri = new float* [n]; // n rows

    for (int i=0; i < n; i++) {
        tri[i] = new float [i];
    }

    return tri;
}
```

Memory used:

n rows

$1+2+\dots+n = n*(n+1)/2$  elements

n pointers of 8 bytes

nb floats:  $n*(n+1)/2$

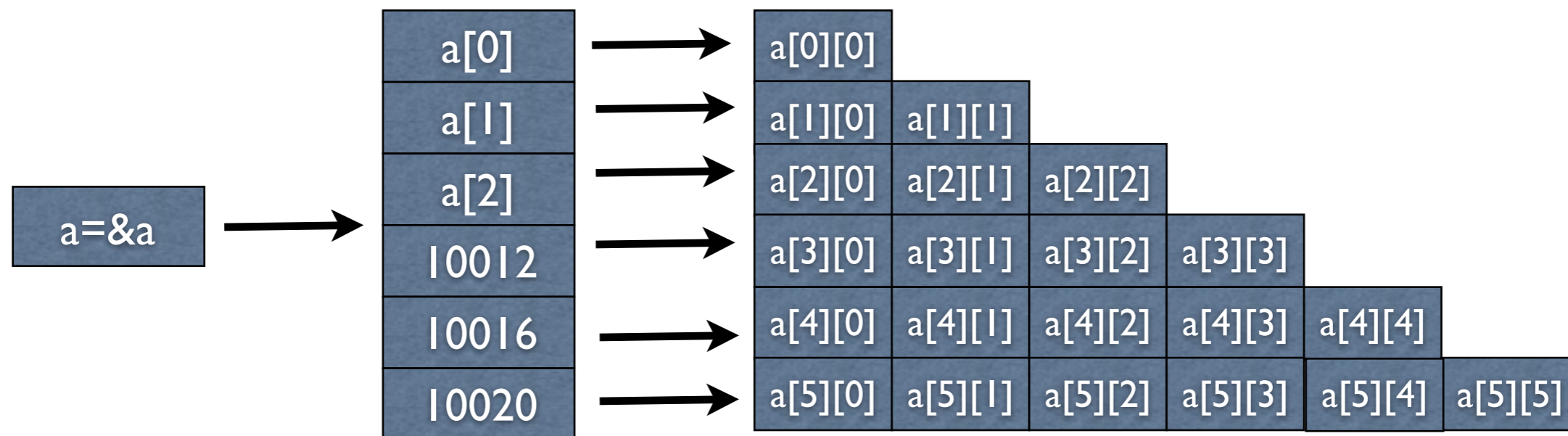
nb pointers: n

total bytes:  $8n+4*n(n+1)/2$   
 $= 2n*n+10n$

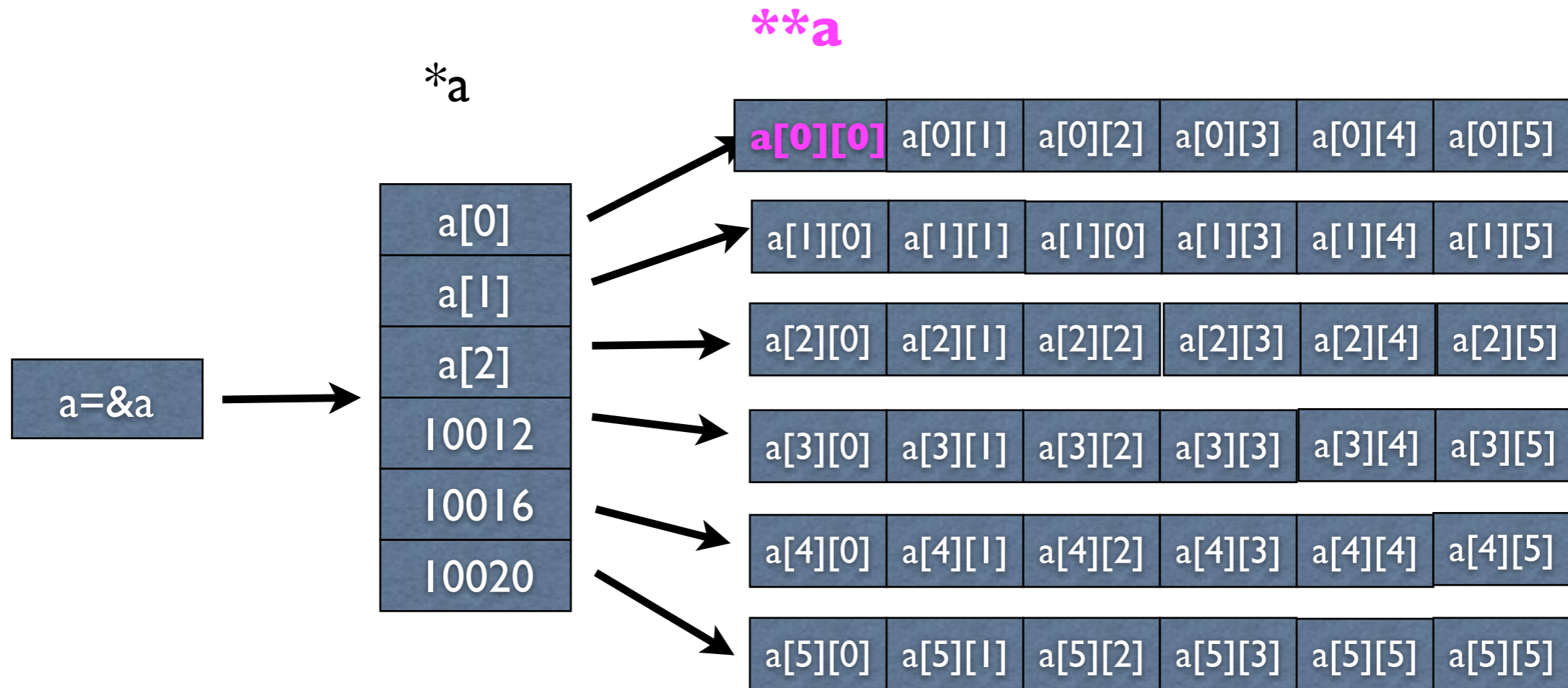
Regular nxn matrix of floats:

uses:  $4*n*n$  (higher memory usage for large n)

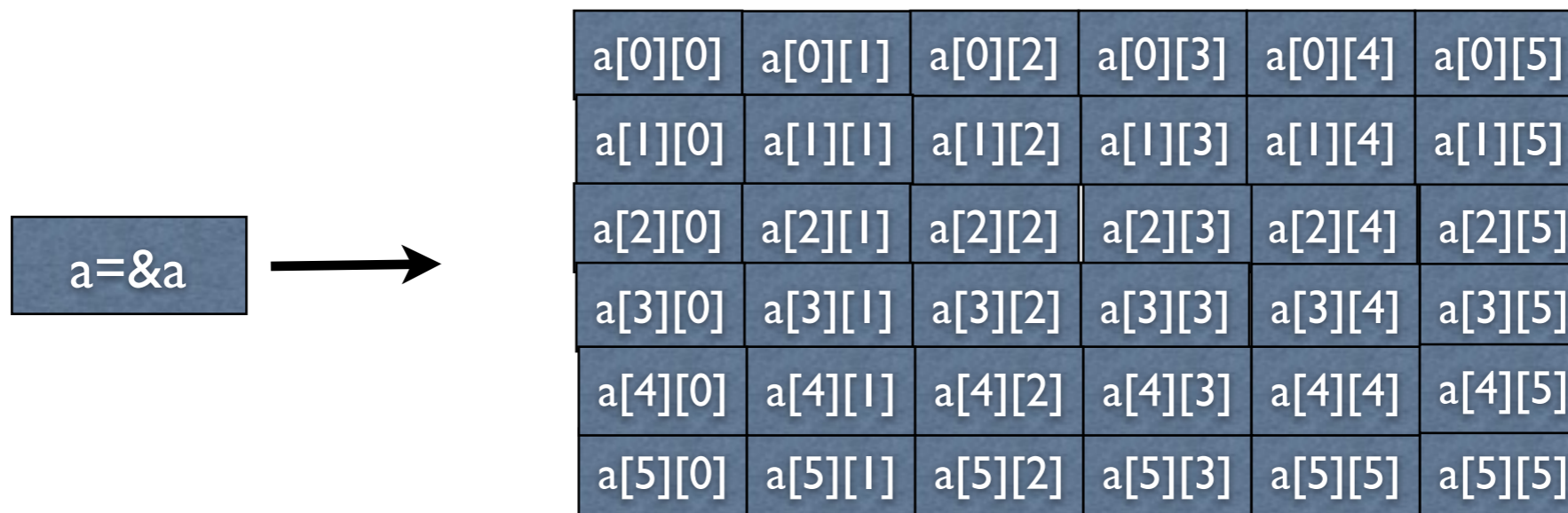
# Triangular Matrix with float\*\* a



# Square 6x6 Matrix with float\*\* a



## Square 6x6 Matrix with float a[6][6]



All elements are contiguous in memory.  
 $\text{addr}(a[5][5]) - \text{addr}(a[0][0]) = (6*6-1)*4$   
(four bytes in every float)



```

int main()
{
    float a[10][10];
    printf("sizeof(a)= %d\n", (int) sizeof(a));

    printf("&a= %ld, a= %ld\n", (long) &a, (long) a);
    printf("&a[0][0]= %ld\n", (long) &a[0][0]);
    printf("a[0]= %ld\n", (long) a[0]);
    printf("&a[0]= %ld\n", (long) &a[0]);

    printf("&a[1]= %ld\n", (long) a[1]);
    printf("&a[1][0]= %ld\n", (long) &a[1][0]);
    printf("&a[2][0]= %ld\n", (long) &a[2][0]);
    printf("&a[0][1]= %ld\n", (long) &a[0][1]);
    printf("&a[0][2]= %ld\n", (long) &a[0][2]);

    printf("&a[0][0]= %ld, &a[9][9]= %ld\n",
        (long) &a[0][0], (long) &a[9][9]);

    int nb_elem = &a[9][9]-&a[0][0] + 1;
    printf("nb_elem= %d\n", nb_elem);

    return 0;
}

```

```

./a.out
sizeof(a)= 400
&a= 140734799801232,
    a= 140734799801232
&a[0][0]= 140734799801232
    a[0]= 140734799801232
&a[0]= 140734799801232
&a[1]= 140734799801272
&a[1][0]= 140734799801272
&a[2][0]= 140734799801312
&a[0][1]= 140734799801236
&a[0][2]= 140734799801240
&a[0][0]= 140734799801232,
&a[9][9]= 140734799801628
nb_elem= 100

```

**All  
identical  
addresses**

40 bytes

40 bytes

4 bytes

**400-4  
=396 bytes**

# Pointer Arithmetic

```
int main()
{
    float* a;
    printf("a= %ld\n", (long) a);

    a = new float [10];
    printf("a= %ld\n", (long) a);

    float* b = a + 1;
    printf("b= %ld\n", (long) b);

    float** c = new float* [10];
    float** d = c + 1;
    printf("c= %ld\n", (long) c);
    printf("d= %ld\n", (long) d);
}
```

./a.out

a= 0

a= 4296016000

b= 4296016004

c= 4296016048

d= 4296016056

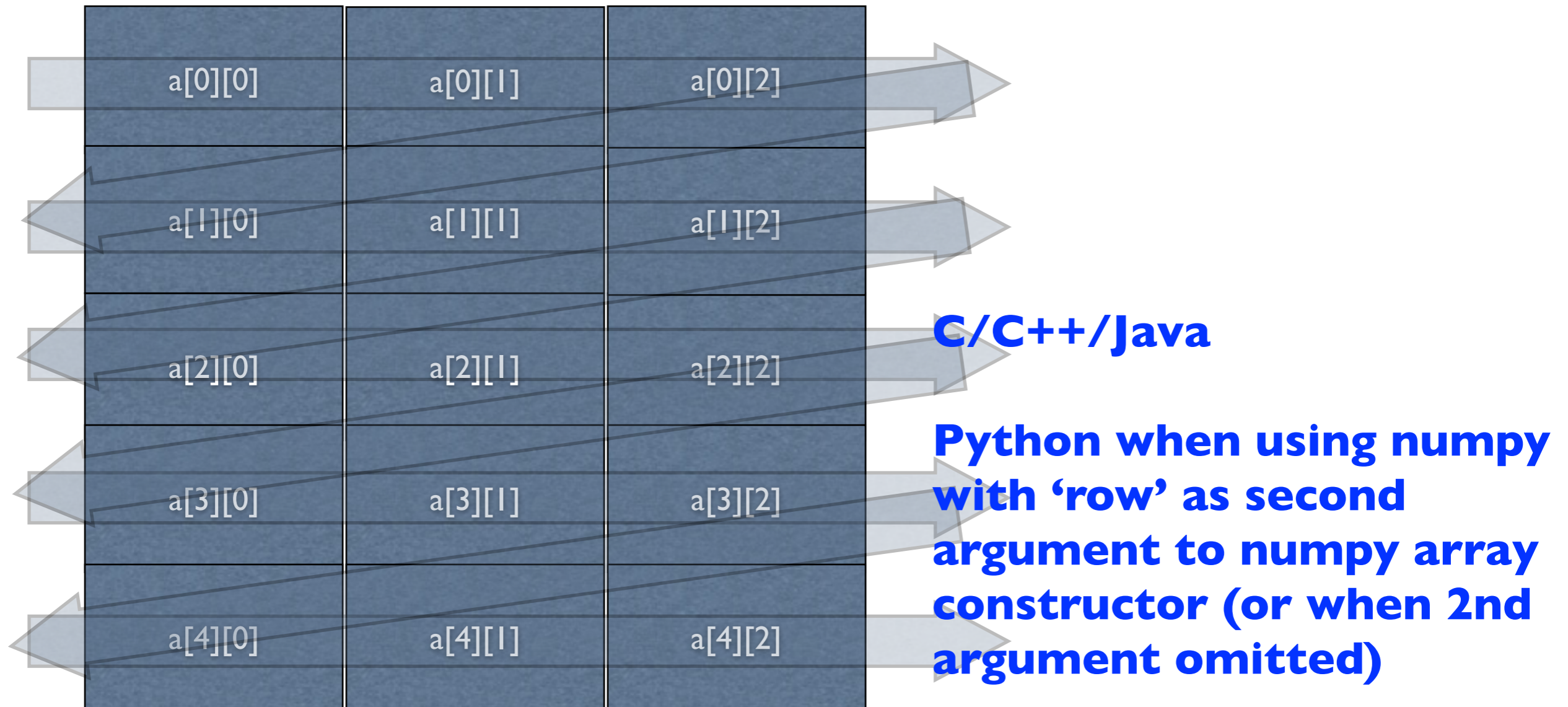
Adding one to a pointer updates the address by the size of the underlying type

# 2D to 1D

- float a[5][3]
  - 15 floats contiguous in memory
  - What order are they stored?
  - a[0][0], a[0][1], ..., a[0][2], a[1][0], a[1][1], a[1][2], a[2][0], a[2][1], a[2][2], a[3][0], ...

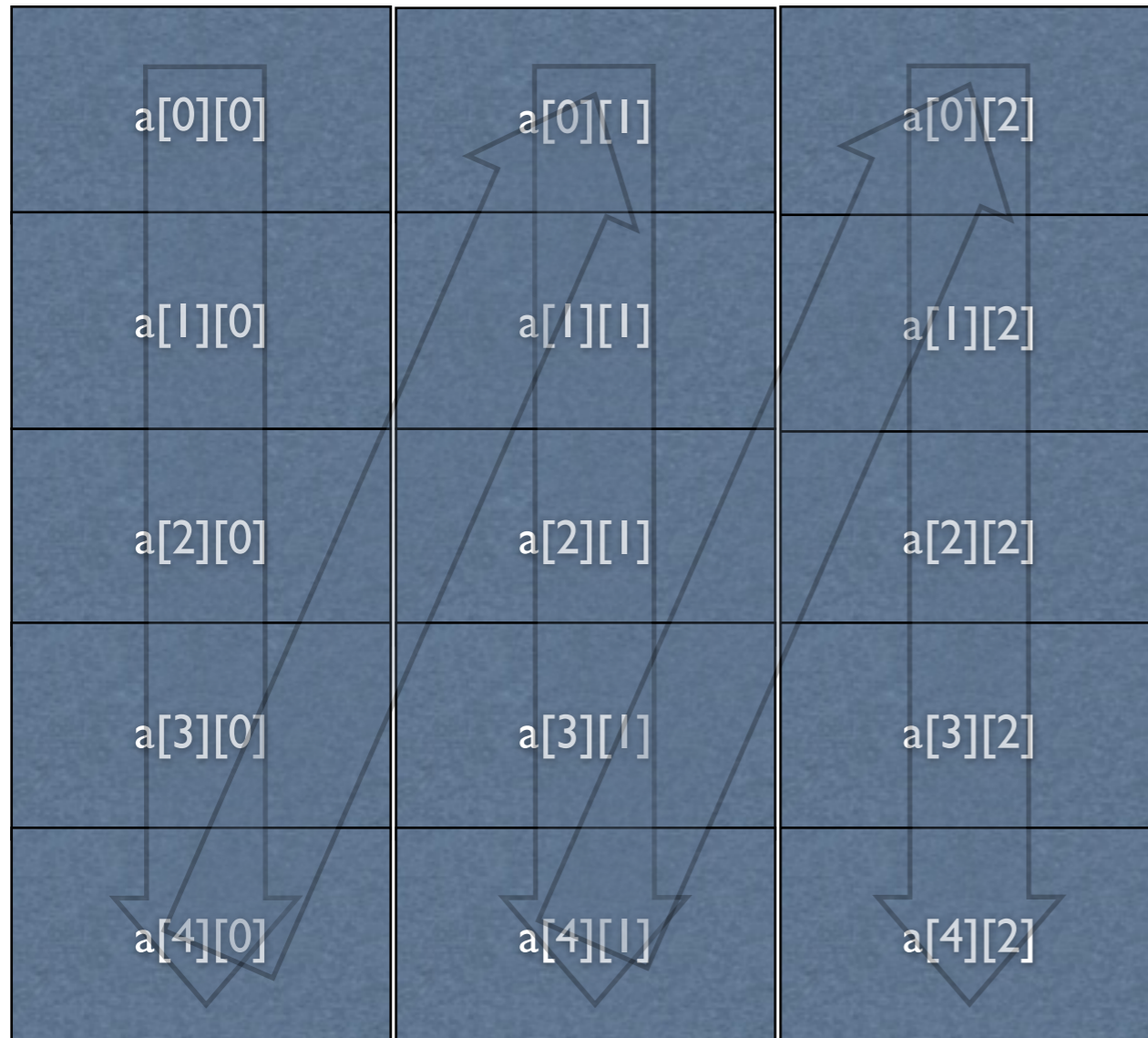
# Rowwise Ordering

- $a[0][0], a[0][1], \dots, a[0][2], a[1][0], a[1][1], a[1][2], a[2][0], a[2][1], a[2][2], a[3][0], \dots$



# Column Ordering

- The ordering is  
 $a[0][0], a[1][0], a[2][0], \dots, a[0][1], a[1][1], \dots$



**Fortran**

**Python when using numpy  
with 'col' as second  
argument to numpy array  
constructor**

# 2D to 1D

- $a[n][m]$  : in C++: 2nd index varies the fastest in memory
- Equivalent 1D array in memory:

$$a[j][i] \text{ equiv } b[i+m*j]$$

$$\text{if } \&a[0][0] == \&b[0]$$

# 3D to 1D

- $a[n][m][p]$  : in C++: 3rd index varies the fastest in memory
- Equivalent 1D array in memory:

$$a[k][j][i] \text{ equiv } b[i+p*j+p*m*k]$$

$$\text{if } \&a[0][0][0] == \&b[0][0]$$

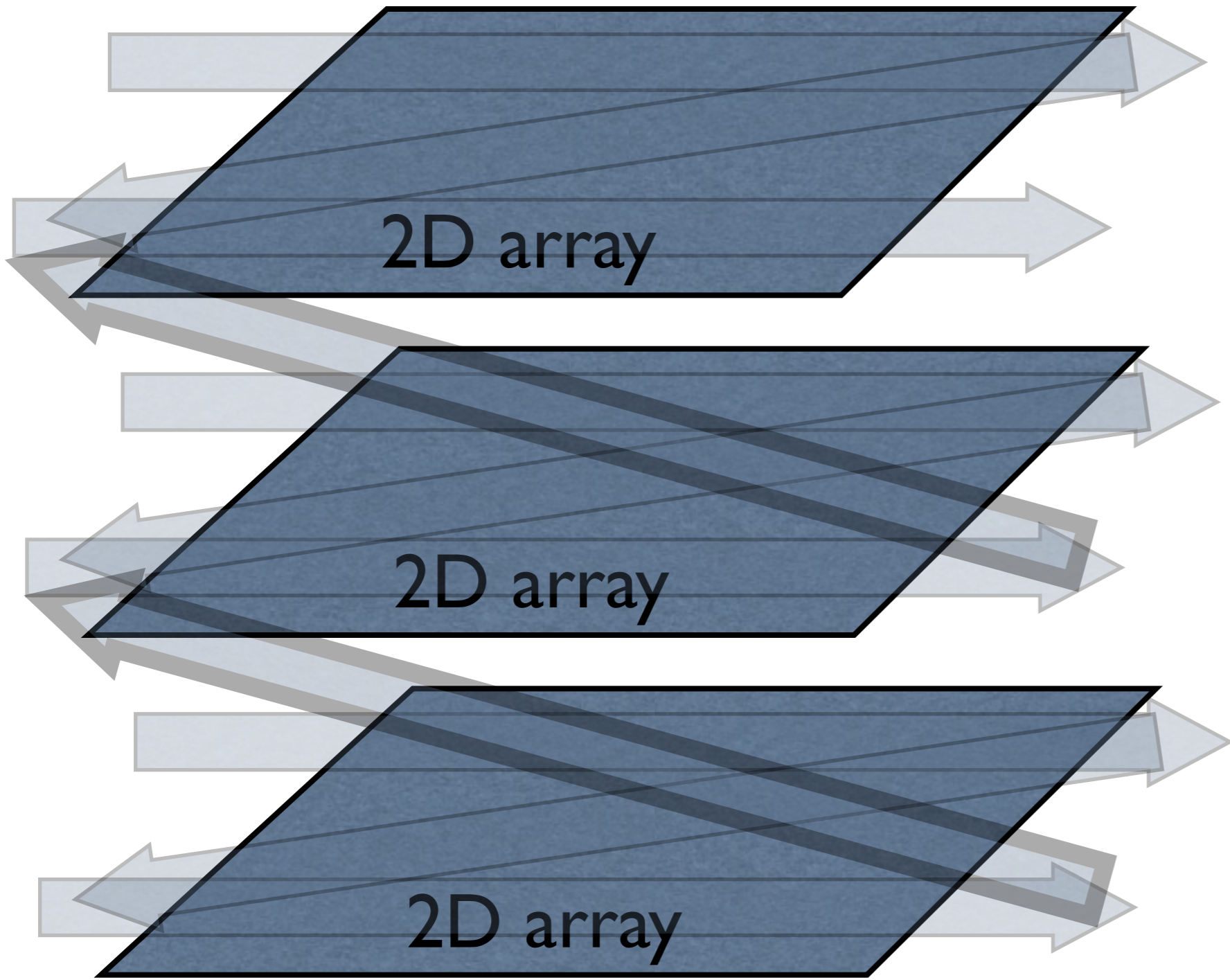
a[m][n][k]

2D array

2D array

2D array

row  
column  
plane





# Pointers and Functions

- pointers can be used as arguments
- pointers can be returned from functions
- Treat each in turn

# Pointer as argument

Memory allocated in **called method**

```
void change(float** p, float* q)
{
    *p = new float [10];
    (*p)[0] = 5;

    q = new float [10];
    q[0] = 5;
}
//-----
int main()
{
    float* p;
    float* q;
    change(&p, q);

    // p[0]=5, q[0] not allocated
    printf("p[0]= %f, q[0]= %f\n", p[0], q[0]);
}
```

Arguments are all passed by value.  
Interpretation: p and q are passed by value.

This value is the same before and after the call.

The address is passed by value.  
Therefore, the method *cannot* change this address

While q changes in the function, **the value is not returned** to the main program.

# Pointer as argument

Memory allocated in **calling method**

```
void change(float* p)
{
    p[0] = 5.;
}
//-----
int main()
{
    float* p = new float[10];
    p[0] = 3.;
    change(p);

    // pointer value has changed to 5
    printf("p[0]= %f\n", p[0]);
}
```

# Returning a pointer

```
//-----  
float* change()  
{  
    float* p = new float [10];  
    p[0] = 5;  
    return p;  
}  
//-----  
int main()  
{  
    float* p = change();  
  
    // pointer value changes to 5 in the function  
    printf("p[0]= %f\n", p[0]); // prints 5  
}  
//-----
```

# returning\_static\_array.cpp

## AVOID!!

```
//-----  
float* change()  
{  
    float p[10];  
    p[0] = 5;  
    return p;  
}  
//-----  
int main()  
{  
    float* p = change();  
  
    // pointer value changes to 5 in the function  
    printf("p[0]= %f\n", p[0]);  
}  
//-----
```

returning\_static\_array.cpp: In function  
'float\* change()':  
returning\_static\_array.cpp:6: warning:  
**address of local variable 'p'**  
**returned**

p is deallocated before the function  
returns