FUNCTIONS IN C

Objectives

To create and access multidimensional arrays

To learn how dynamic arrays work

MULTIDIMENSIONAL ARRAYS

Arrays with more than one dimension.

```
<data_type> <name>[size1][size2]…[sizeN];
```

```
//two-dimensional array
//of integers
int table[3][4];
```

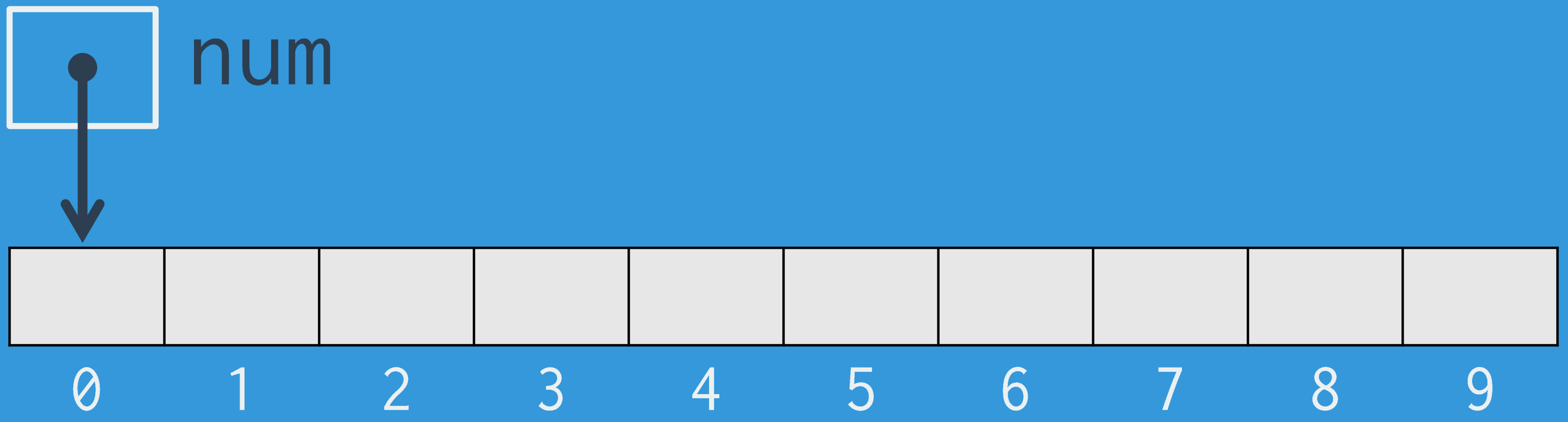
```
//3d array of float
float rgb[255][255][255];
```

```
//initializing a 2d array
int table[3][4] =
  { {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
  };
```

1	2	3	4
5	6	7	8
9	10	11	12

2D arrays in the memory

In 1D-arrays,

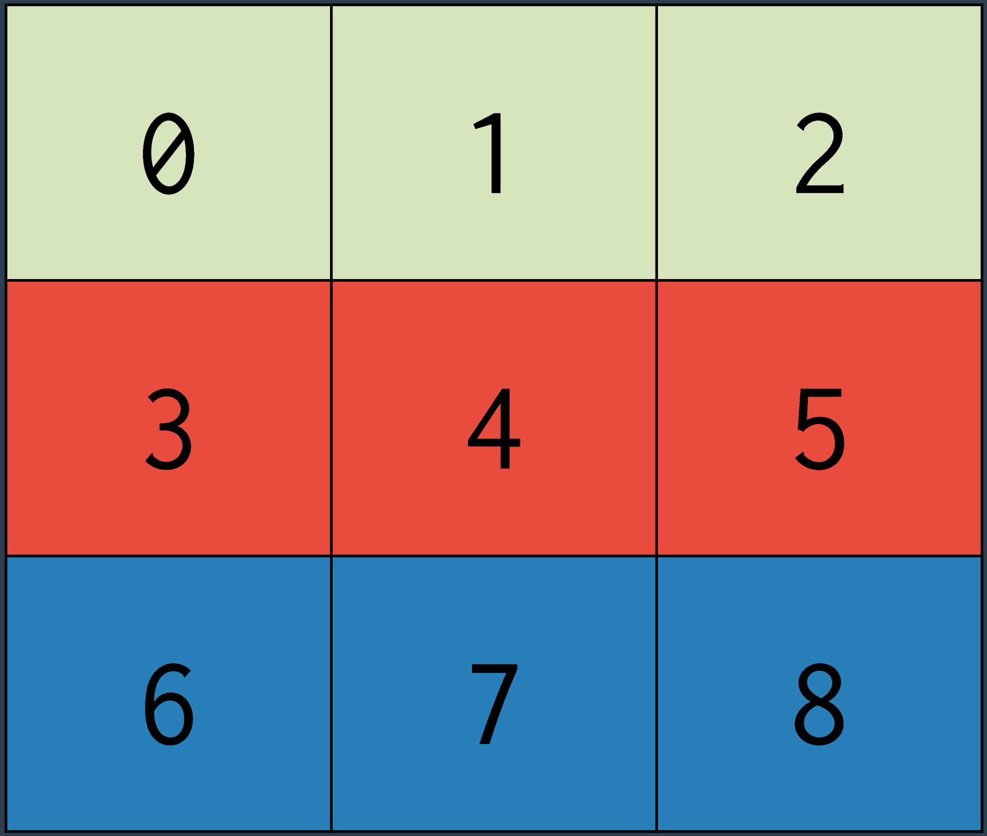


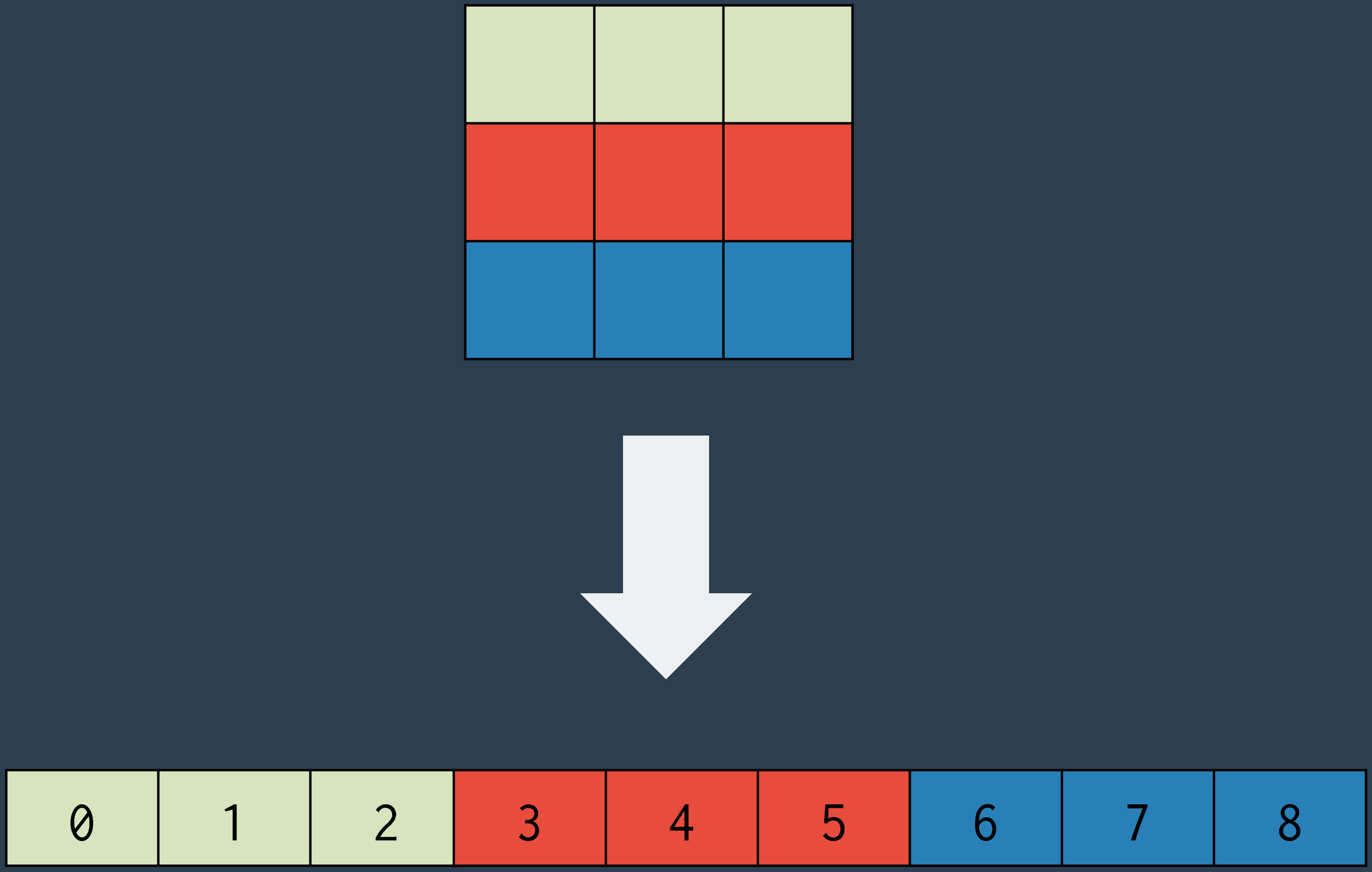
The variable name of a 2D array can be treated as a pointer to an array of pointers.

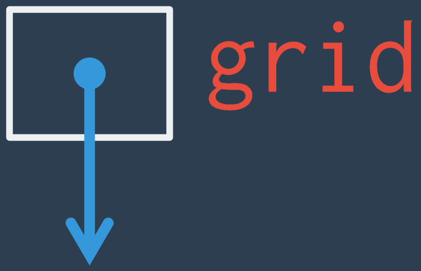
And each pointer (in the array) holds the address of the first element.

```
int grid[3][3];
```

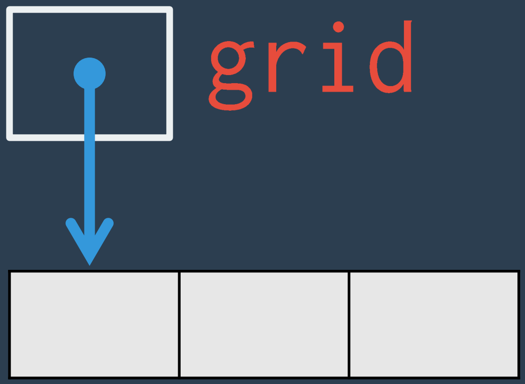
0	1	2
3	4	5
6	7	8


0
1
2
3
4
5
6
7
8

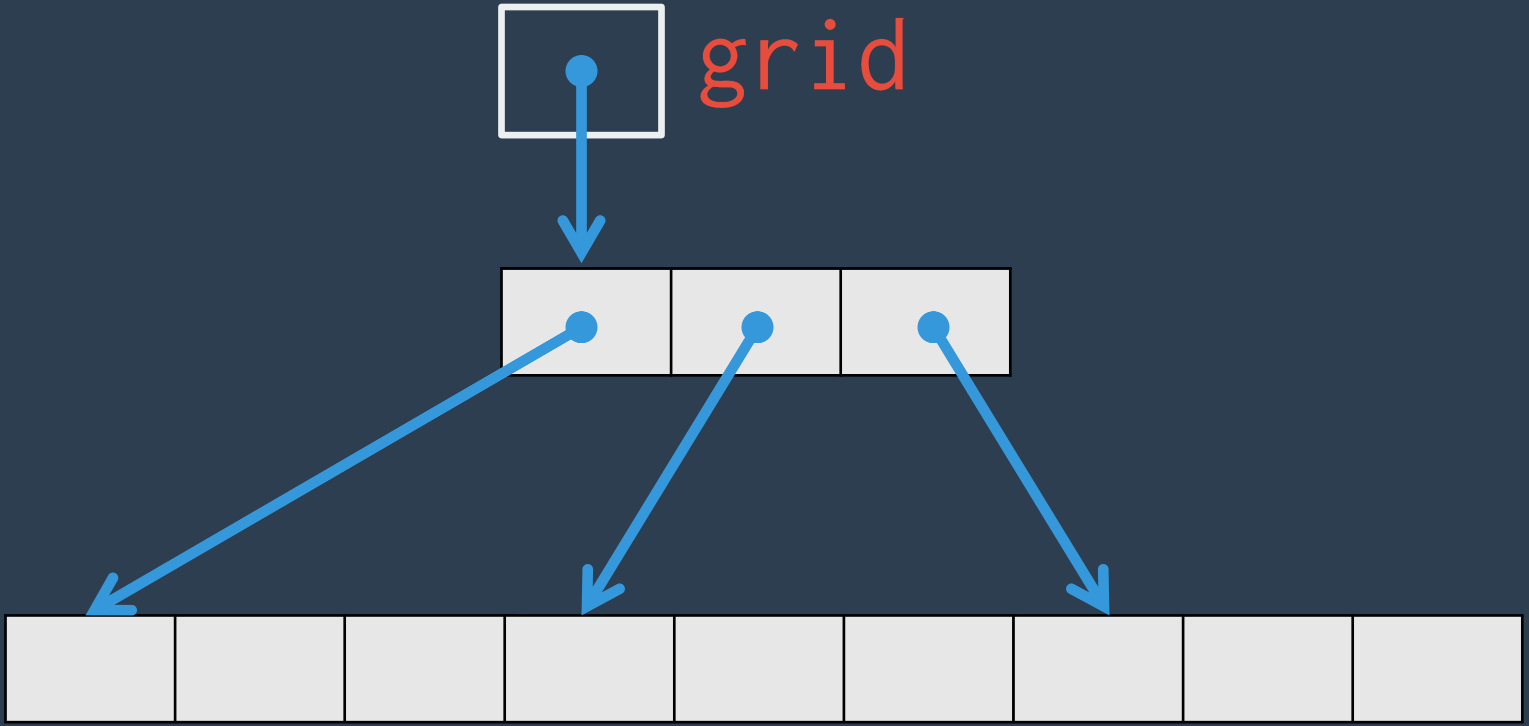

0
1
2
3
4
5
6
7
8



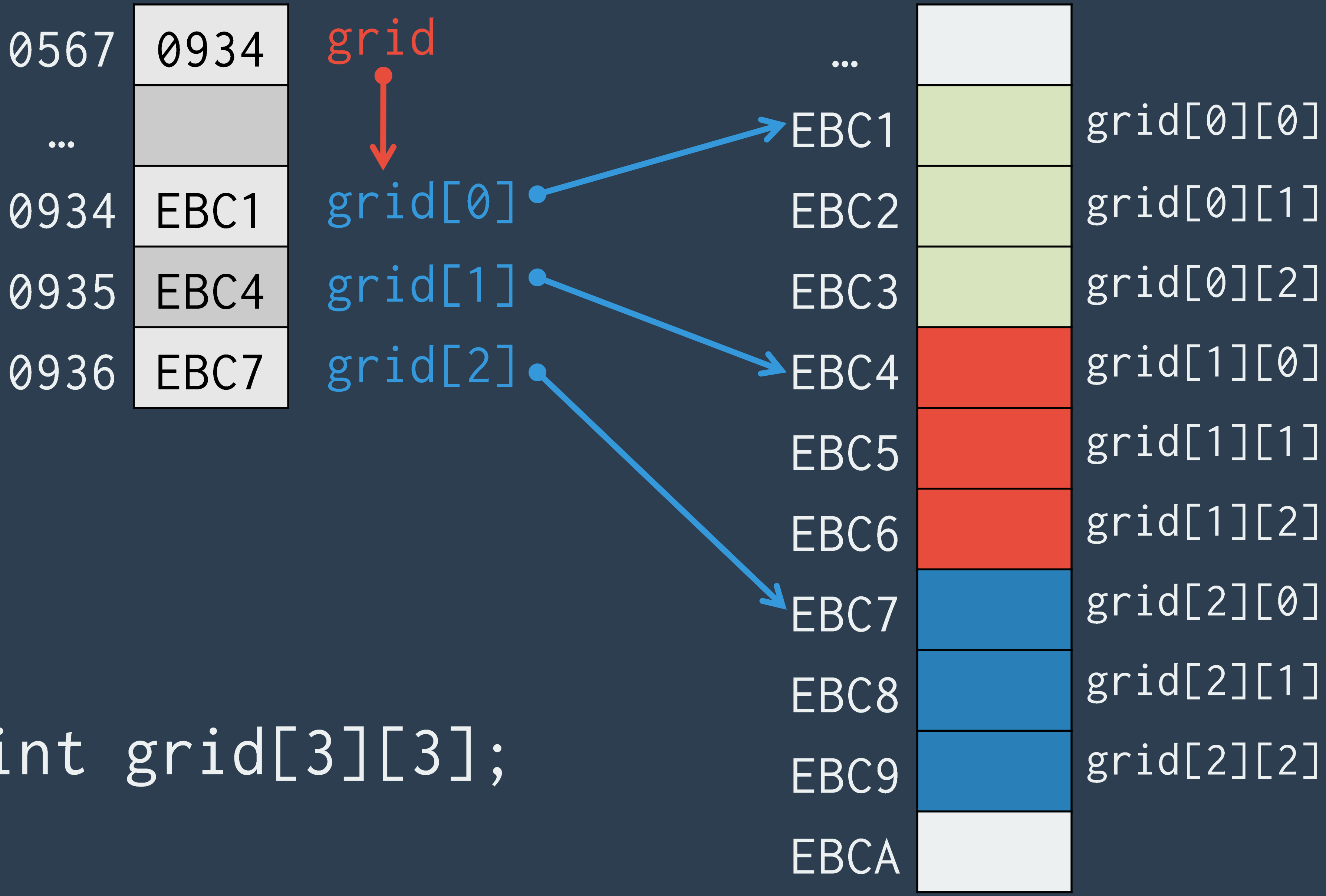
grid is the 2D array variable name



and it points to an array of pointers



which point to arrays (via the first element)


0567
0934
grid
…
0934
EBC1
grid[0]
0935
EBC4
grid[1]
0936
EBC7
grid[2]
…
EBC1
EBC2
EBC3
EBC4
EBC5
EBC6
EBC7
EBC8
EBC9
EBCA
grid[0][0]
grid[0][1]
grid[0][2]
grid[1][0]
grid[1][1]
grid[1][2]
grid[2][0]
grid[2][1]
grid[2][2]

```
int grid[3][3];
```

Accessing Multidimensional Arrays

Indexing

+

Pointer arithmetic

Indexing is just like in 1D arrays.

```
grid[1][0] = 27;
```

In general,

$\texttt{<arr\_name>[}i_1\texttt{][}i_2\texttt{]}\ldots\texttt{[}i_n\texttt{]}$

Pointer arithmetic is also similar.

```
//grid[1][0] = 27;
*(*(grid+1)+0) = 27;
```

In general,

$$*(\ldots*(*(\texttt{<arr\_name>}+i_1)+i_2)+\ldots+i_n)$$

Multidimensional arrays as parameters

The name of the array is specified as the actual parameter.

```
int main()
{
    int m[3][4];
    initialize(m);
}
```

For formal parameters,

Specify all the sizes…

```c
void initialize(int m[3][4])
{
    int i, j;
    for(i=0;i<3;i++)
        for(j=0;j<4; j++)
            m[i][j] = i+j;
}
```

… or specify the sizes
except for the first.

```c
void initialize(int m[][4])
{
    int i, j;
    for(i=0;i<3;i++)
        for(j=0;j<4; j++)
            m[i][j] = i+j;
}
```

This is done so that the compiler knows the “depths” of each additional dimension.

DYNAMIC ARRAYS

```
//arrays declared like this
//are called STATIC ARRAYS
int grid[3][3];
```

What are DYNAMIC ARRAYS?

These are arrays that are allocated in the memory at runtime.

The size of the array can be set during the execution of the program.

Dynamic arrays are allocated using:

Pointers

+

malloc()

malloc()

malloc() is used to allocate a specific amount of memory during execution of a program.

`malloc() is in stdlib.h`

```
void * malloc(size_t size);
void * malloc(size_t size);
```

An unsigned int returned by the operator sizeof.

```
void * malloc(size_t size);
```

If successful, a pointer (to the memory location) is returned.

```
void * malloc(size_t size);
```

Otherwise, a NULL pointer is returned.

```
void * malloc(size_t size);
```

The type of pointer returned is always void *.

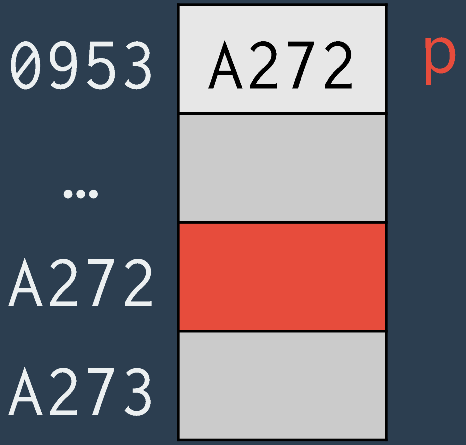
```
void * malloc(size_t size);
```

This pointer can be typecasted to the desired type of data.

```
#include<stdlib.h>
int main()
{
   int *p;
   //allocate one dynamic integer in
   //the memory
   p = (int *) malloc(sizeof(int));
}
```

The newly created dynamic variable has no name or identifier.

Dynamic variables can be accessed using pointers.

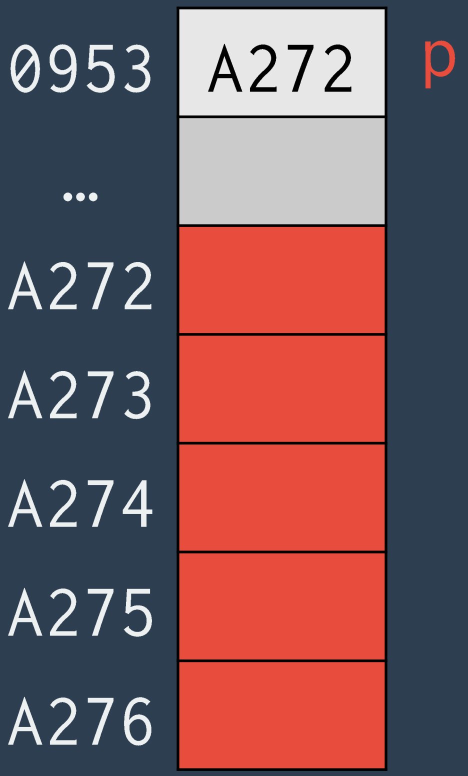


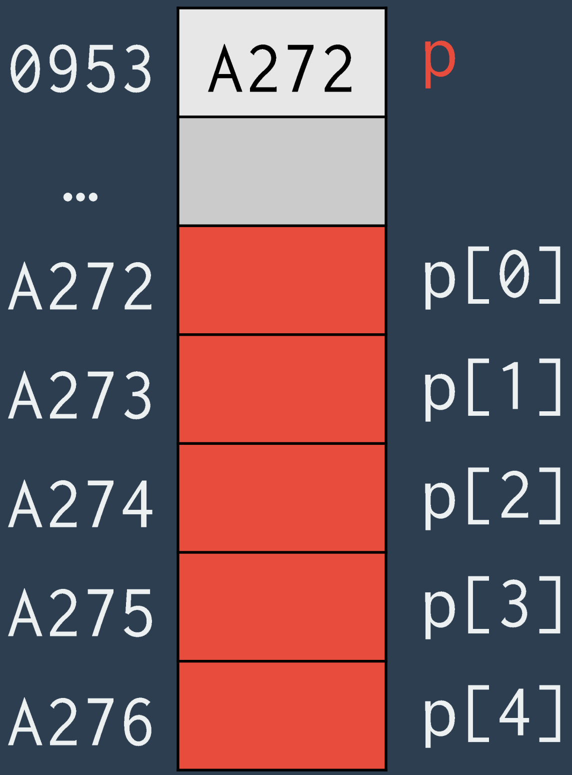
Dynamic 1D arrays

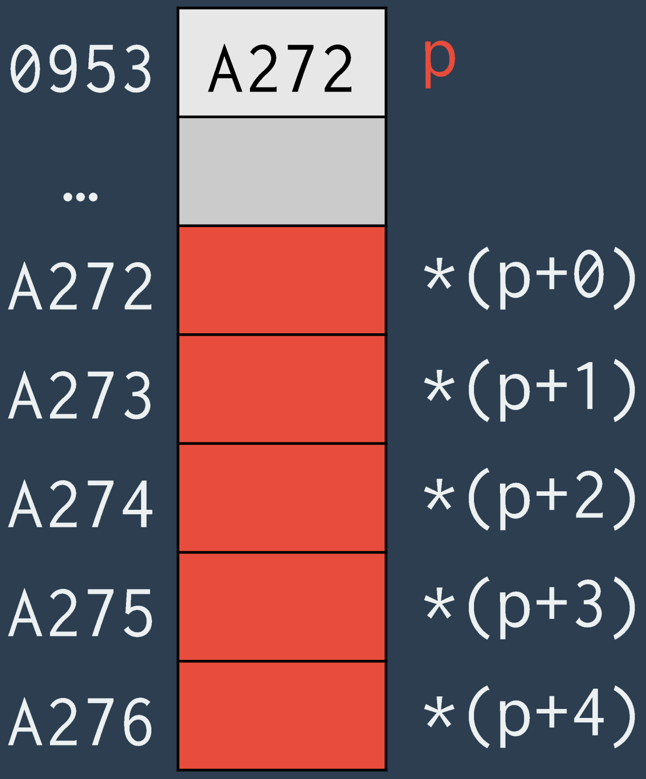
```
#include<stdlib.h>
int main()
{
  int *p, size=5;
  //allocate 5 dynamic integers
  p = (int *) malloc(size * sizeof(int));
}
```

```c
#include<stdlib.h>
int main()
{
  int *p, size=5;
  //allocate 5 dynamic integers
  p = (int *) malloc(size * sizeof(int));
}
```

Reserve memory space for 5 integers.


0953
A272
p
...
A272
A273
A274
A275
A276


0953
A272
p
...
A272
p[0]
A273
p[1]
A274
p[2]
A275
p[3]
A276
p[4]


0953
A272
p
...
A272
*(p+0)
A273
*(p+1)
A274
*(p+2)
A275
*(p+3)
A276
*(p+4)

Dynamic MultiD arrays

Dynamic multidimensional arrays are allocated using pointers to pointers.

A dynamic 2d array uses a pointer to a pointer…

…, a dynamic 3d array uses a pointer to a pointer to a pointer, and so on.

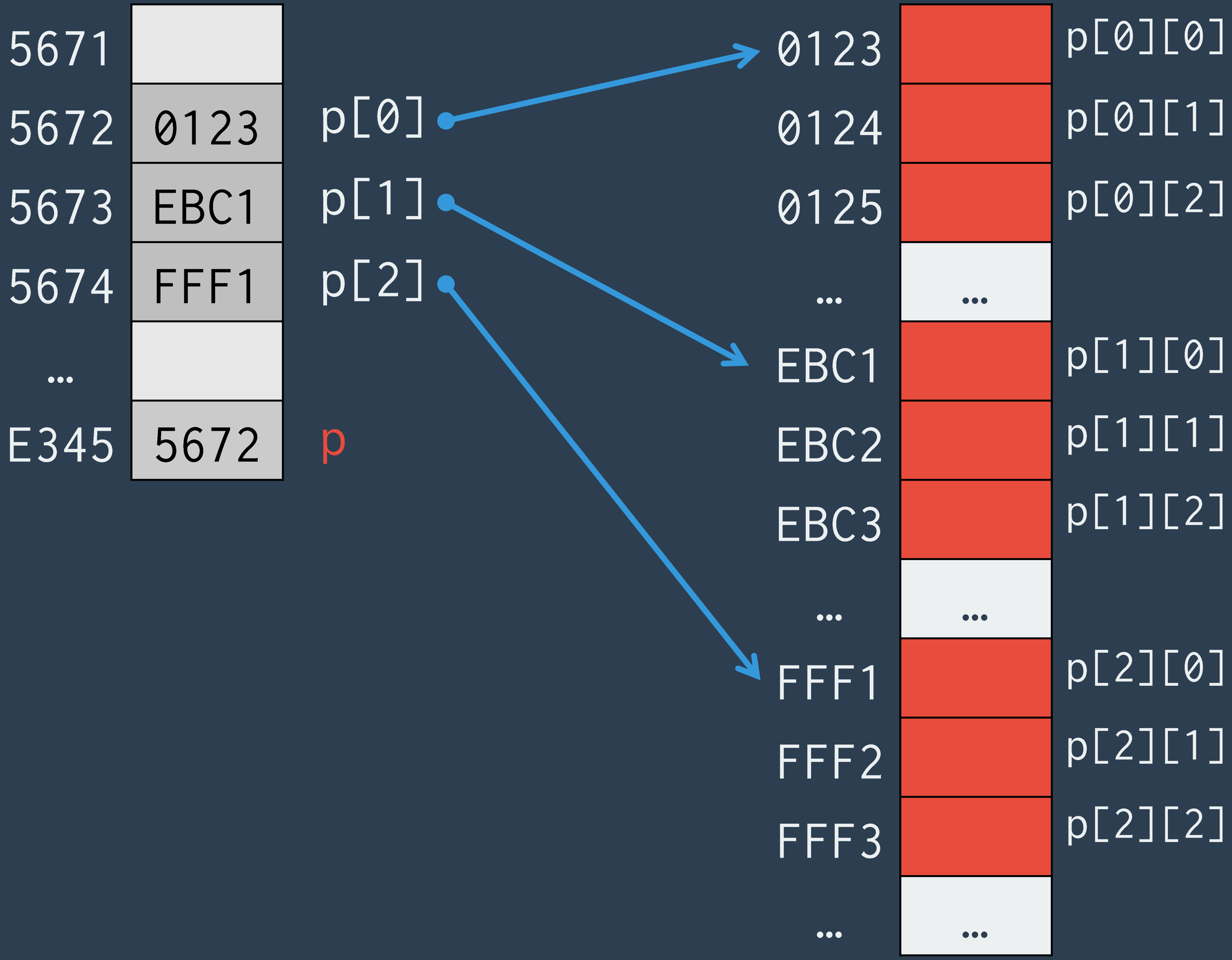
What happens when a multiD array is ALLOCATED DYNAMICALLY?

Let’s declare a 2D array dynamically.

```c
#include<stdlib.h>
#define row 3
#define col 3

int main()
{
  //Declare a pointer to a pointer
  int **p, i;
```

```c
#include<stdlib.h>
#define row 3
#define col 3

int main()
{
  int **p, i;
  //allocate row # of pointers
  //then let p have the address
  p = (int **) malloc(row*sizeof(int *));
```

```c
#include<stdlib.h>
#define row 3
#define col 3

int main()
{
  int **p, i;
  p = (int **) malloc(row*sizeof(int *));

  for(i=0; i<row; i++)
  {
    //allocate space for array elements
    *(p+i) = (int *) malloc(col*sizeof(int));
  }
}
```



5671
5672
0123
5673
EBC1
5674
FFF1
...
E345
5672
p
p[0]
p[1]
p[2]
0123
0124
0125
...
...
EBC1
EBC2
EBC3
...
...
FFF1
FFF2
FFF3
...
...
p[0][0]
p[0][1]
p[0][2]
p[1][0]
p[1][1]
p[1][2]
p[2][0]
p[2][1]
p[2][2]

To pass a dynamic array to a function, the name of the pointer is specified as actual parameter (w/ corresponding formal parameter).

PROBLEM 4.

How many memory spaces does the following occupy?

1. A dynamic 1D array of size 3.
2. A dynamic 2D array with five rows and four columns.
3. A dynamic 4x4x4 array.

Peculiarities of static C arrays

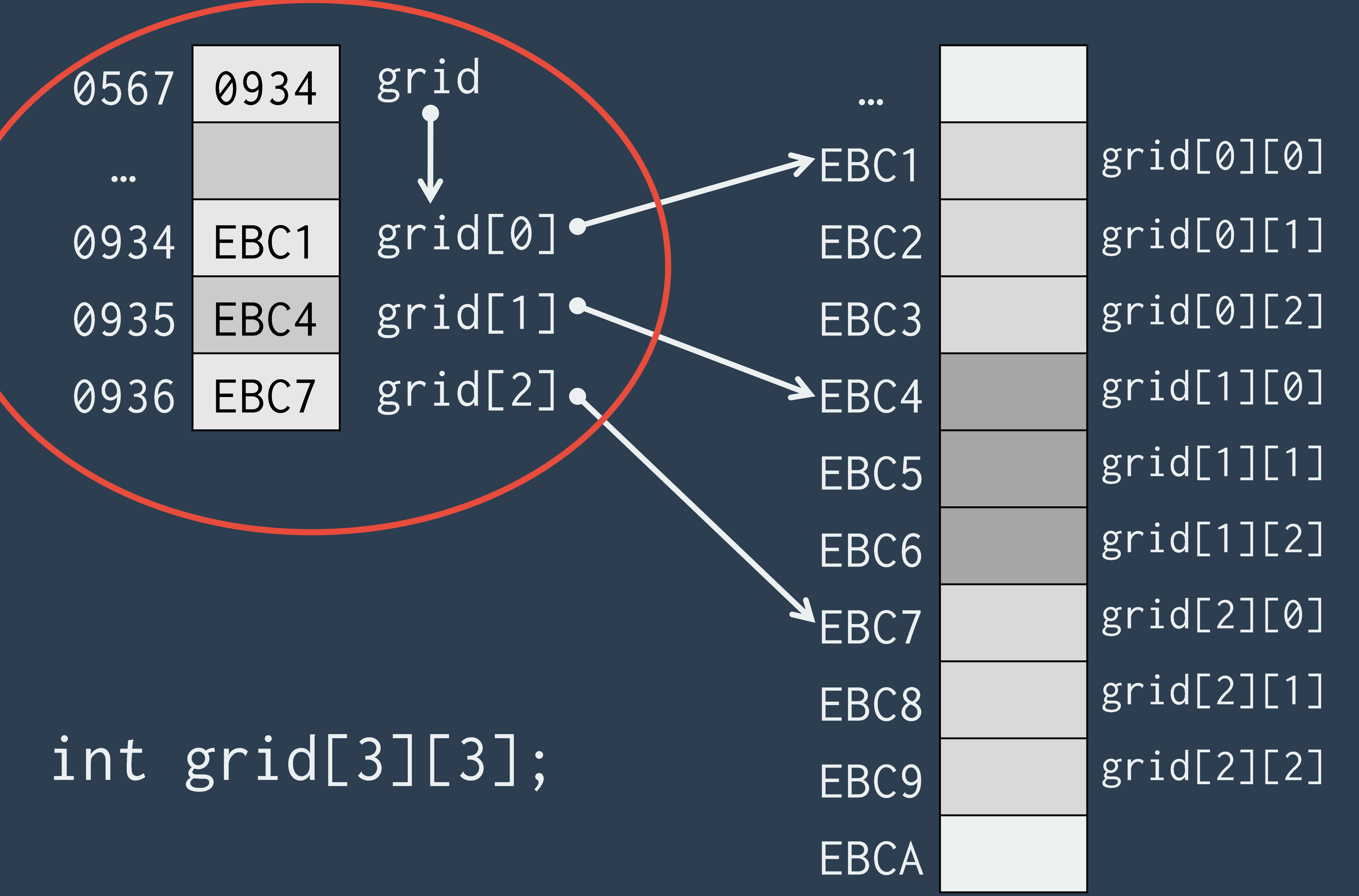
```c
int main()
{
  int a[3][3], **p2p;

  p2p = a; //is this valid?
}
```

```
int main()
{
  int a[3][3], **p2p;

  //will produce a warning
  p2p = a;
  //segmentation fault
  printf(“%d”, **p2p);
}
```

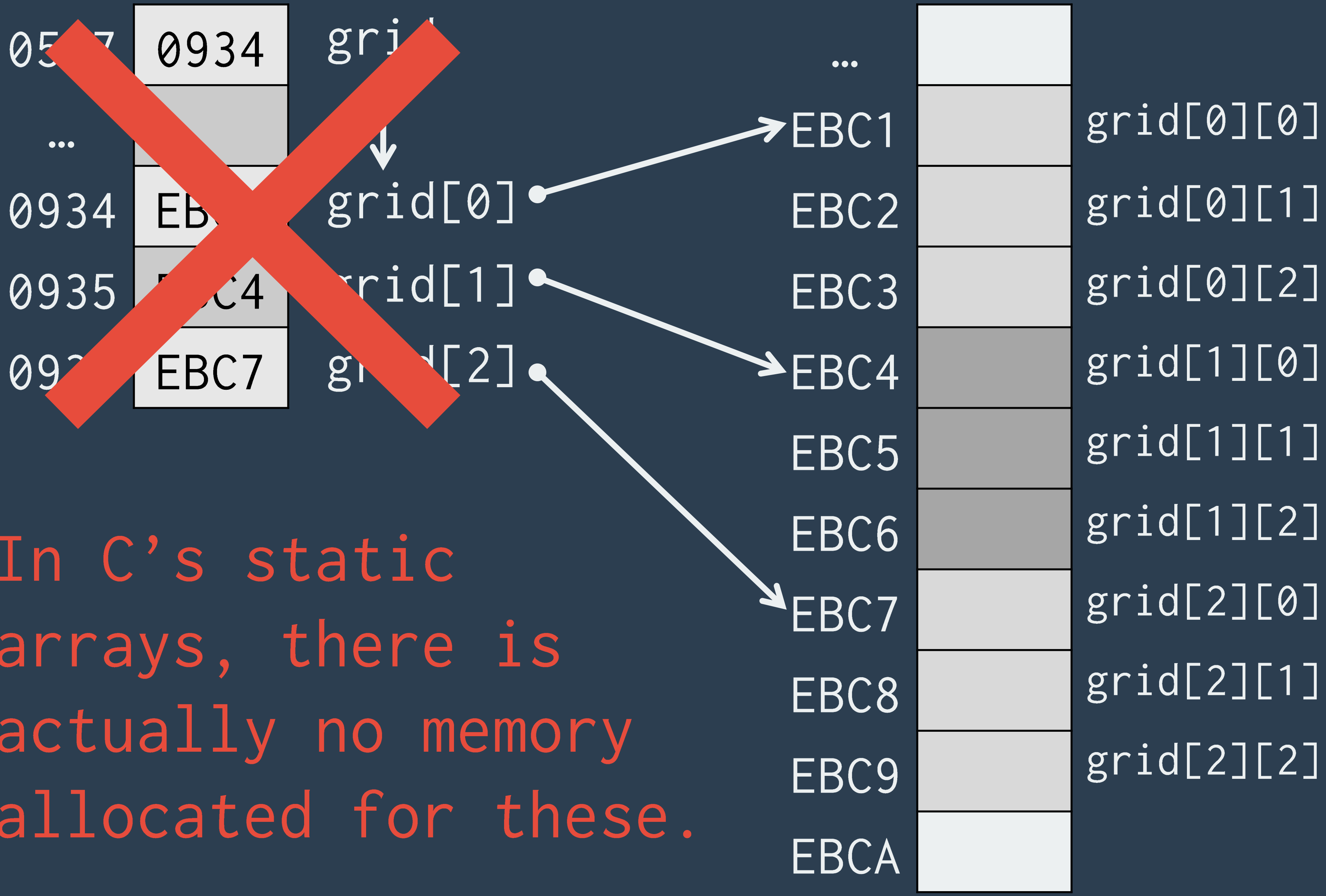
```
int main()
{
  int b[4];

  //FOR STATIC ARRAYS
  printf(“%p\n”, b);
  printf(“%p\n”, &b);

}
```

```
int main()
{
  int a[3][3];

  //FOR STATIC ARRAYS
  printf(“%p\n”, a);
  printf(“%p\n”, &a);
  printf(“%p\n”, a[0]);
  printf(“%p\n”, &a[0]);
  printf(“%p\n”, &a[0][0]);
}
```

In C, a static 2D array is not the same as an array of pointers to 1D arrays.


0567
0934
grid
…
0934
EBC1
grid[0]
0935
EBC4
grid[1]
0936
EBC7
grid[2]
…
EBC1
EBC2
EBC3
EBC4
EBC5
EBC6
EBC7
EBC8
EBC9
EBCA
grid[0][0]
grid[0][1]
grid[0][2]
grid[1][0]
grid[1][1]
grid[1][2]
grid[2][0]
grid[2][1]
grid[2][2]

```
int grid[3][3];
```



In C's static arrays, there is actually no memory allocated for these.

```
int main()
{
   int a[3][3];

   //FOR STATIC ARRAYS
   printf("%p\n", a);
   printf("%p\n", &a);
   printf("%p\n", a[0]);
   printf("%p\n", &a[0]);
   printf("%p\n", &a[0][0]);
}
```

A C compiler treats these automatically.

What does this mean for you?

Nothing. Yet.