Tirgul4 - Agenda

- Multi-dimensional arrays
- Pointer arithmetic
- Compilation & linkage

Multi-dimensional arrays

- Static arrays
- Semi-dynamic arrays
- Fully-dynamic arrays

Access: arr[i][j]

So what is the difference? How it is stored in memory.

Static arrays: int arr[4][8];

- Continuous memory
- Efficient: one memory access to read an index

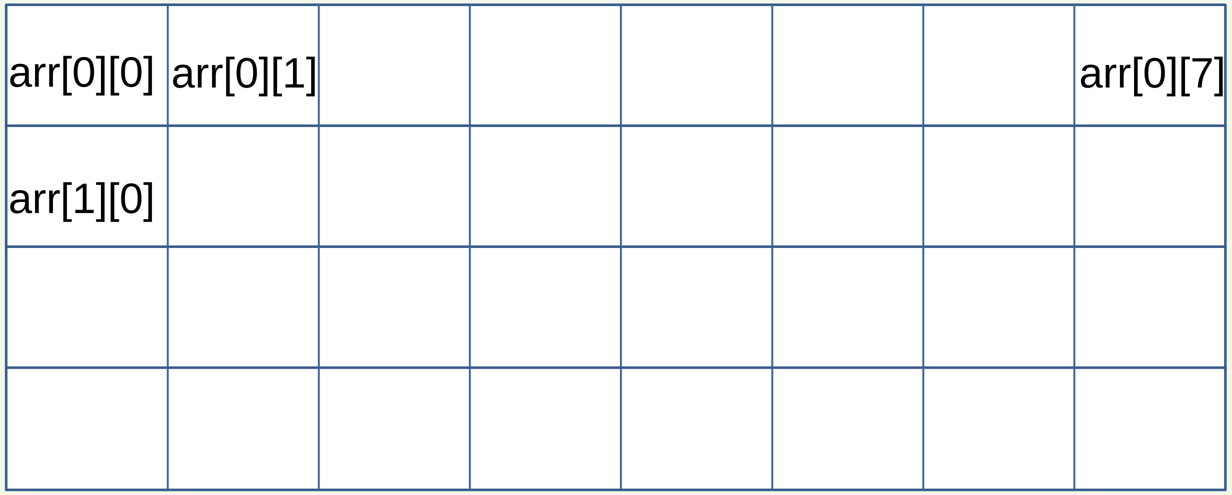
but it is not always possible to use it…

- Size must be known at compile time

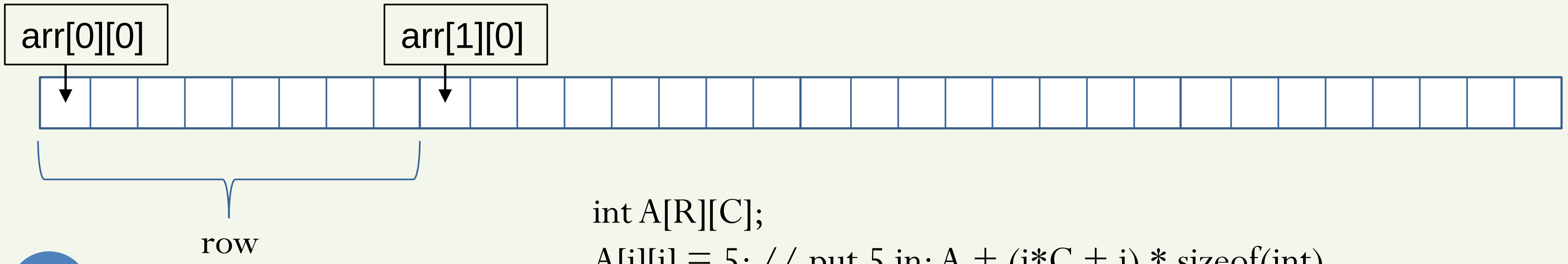
sizeof(arr) = 4*8*sizeof(int)

Static arrays: int arr[4][8];

STACK



How it actually seems:



int A[R][C];
A[i][j] = 5; // put 5 in: A + (i*C + j) * sizeof(int)

Semi-dynamic arrays: int *arr[4];

- Size of each row might be different
- Less efficient: two memory access to read an index

but here too… It is not always possible to use it

- Number of rows must be known at compile time

Read the expression from right to left: int *arr[4];

1. read the variable name (arr)
2. right: [4] means array of 4.

4 cells of which type?

3. left: (int *) means pointer to int

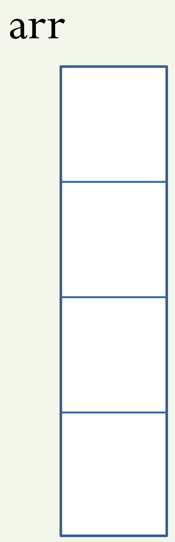
arr is an array of 4 **pointers** to int.

sizeof(arr) = 4 * sizeof(int*)
sizeof(*arr) = sizeof(int*)

Semi-dynamic arrays

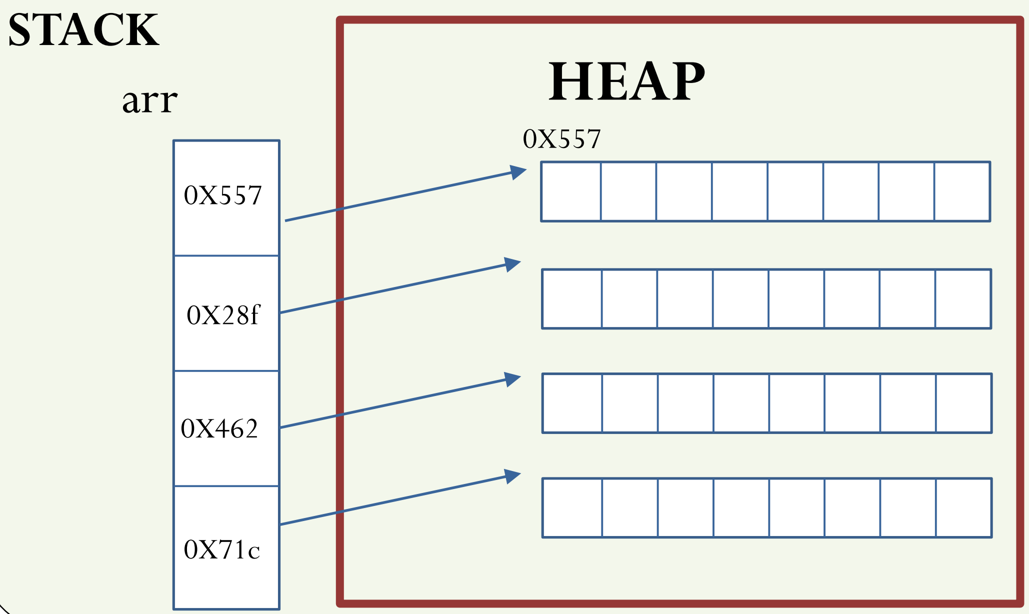
```
int *pa[4]; // allocates memory for 4 pointers
```

STACK

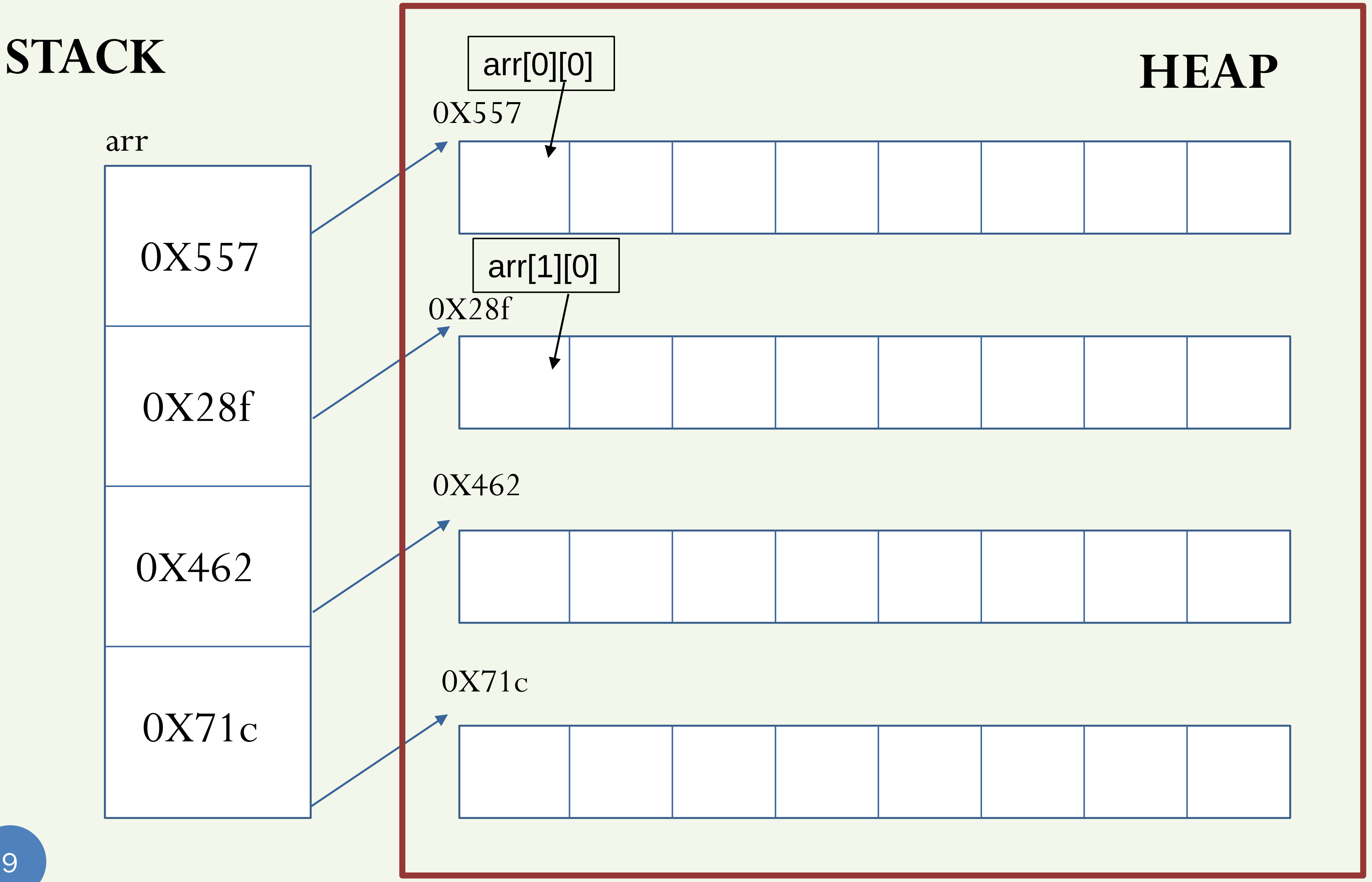


Semi-dynamic arrays: int *arr[4];

```
int *pa[4]; // allocates memory for 4 pointers
for (int i=0; i<4; i++)
{
    pa[i] = (int*) malloc( 8*sizeof(int) );
}
```



Semi-dynamic arrays: int *arr[4];



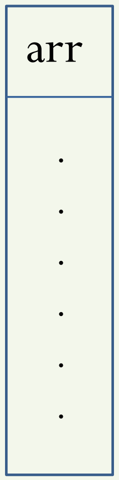
Fully-dynamic arrays: int ** arr;

- Size may be unknown at compile-time
- Even less efficient: three memory access to read an index

fully-dynamic arrays: int **arr;

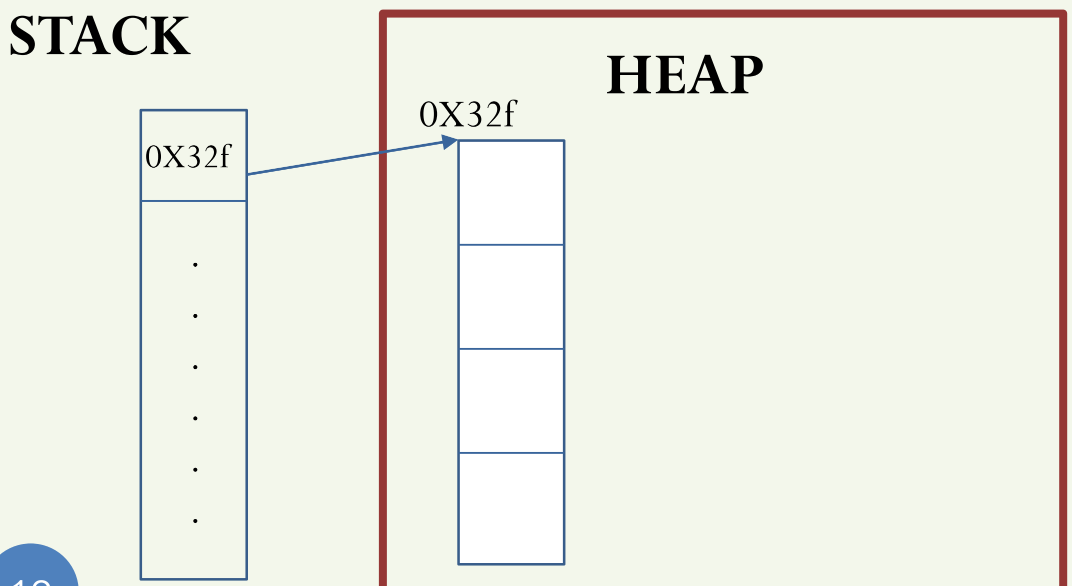
```
int ** arr;
```

STACK



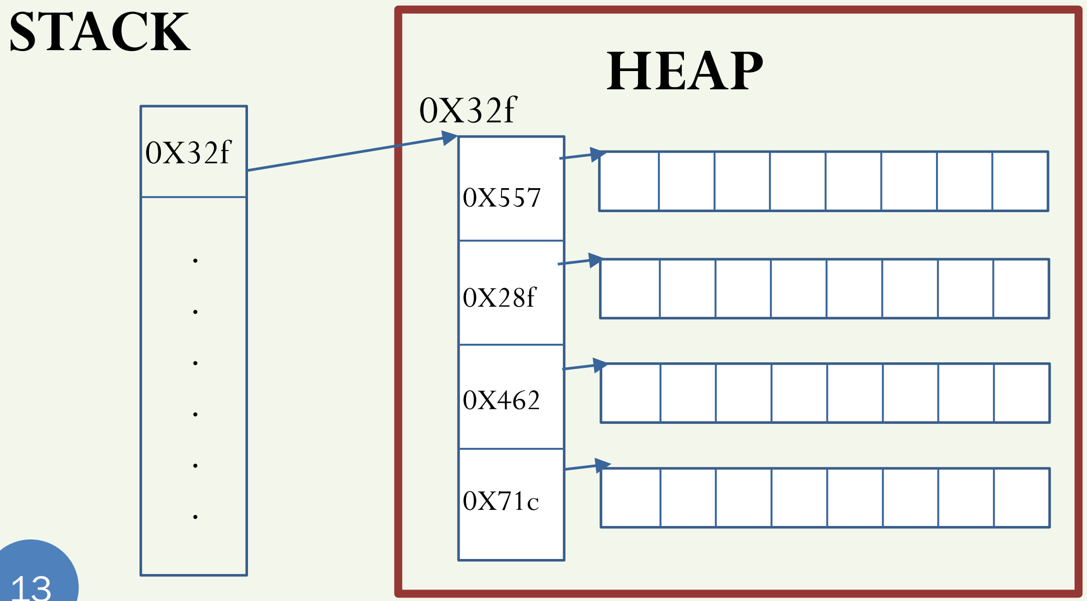
fully-dynamic arrays: int **arr;

```
int ** arr;
arr = (int**)malloc(4*sizeof(int*));
printf(sizeof(*arr)); \\ sizeof(int*)
```

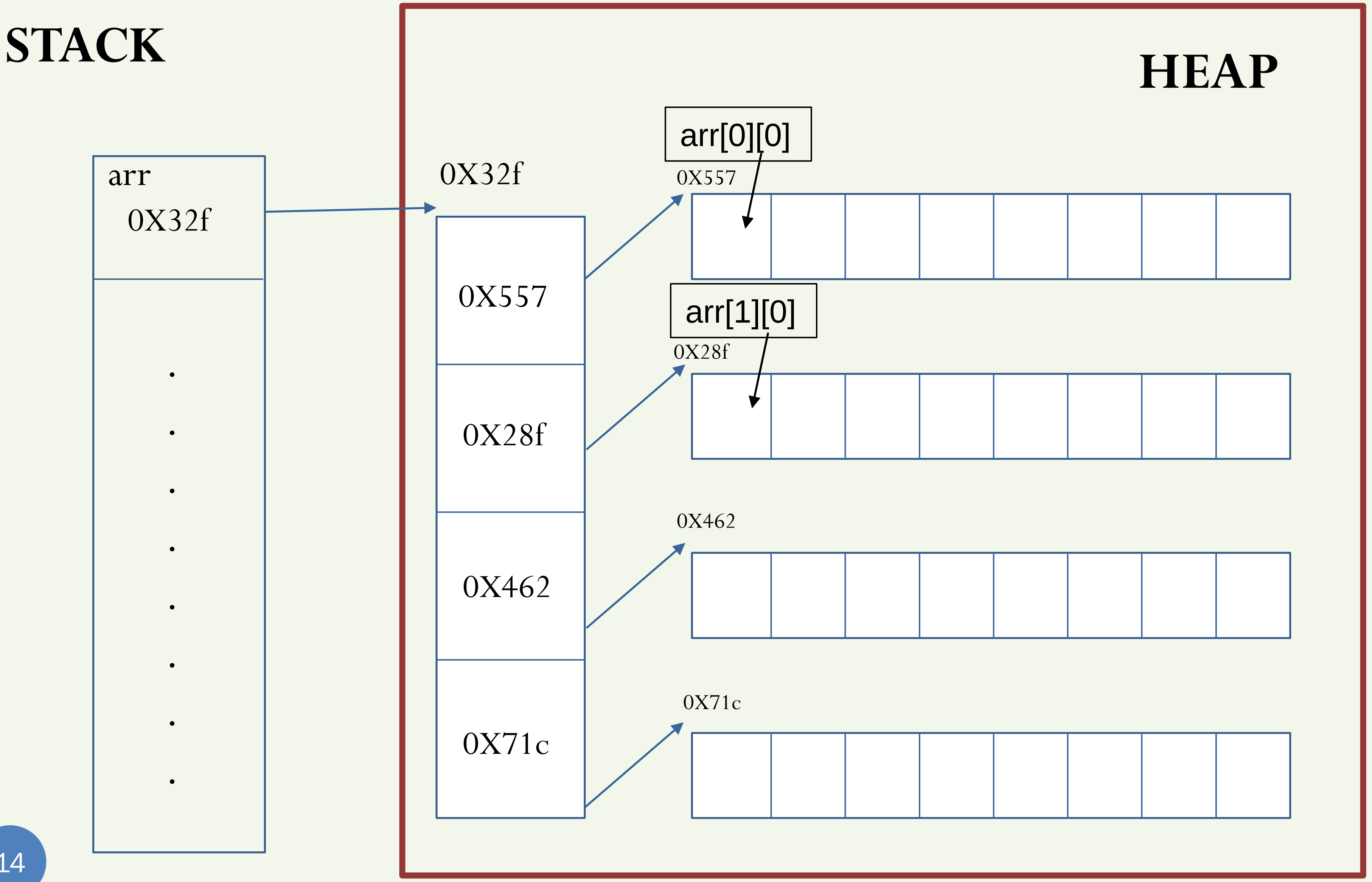


fully-dynamic arrays: int **arr;

```
int ** arr;
arr = (int**)malloc(4*sizeof(int*));
for (i=0; i<4; i++)
{
    arr[i] = (int*)malloc(8*sizeof(int));
}
```



Fully-dynamic arrays: int **arr;

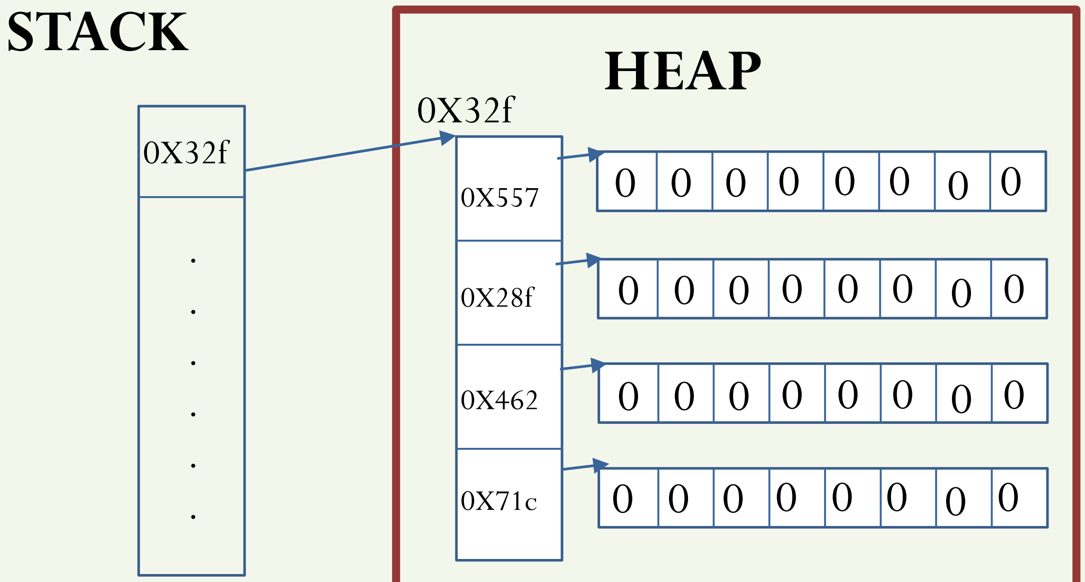


Note:

YES - you allocated the memory, **but** the cells are not initialized yet with corresponding values

fully-dynamic arrays: int **arr;

```
int ** arr;
arr = (int**)malloc(4*sizeof(int*));
for (i=0; i<4; i++)
{
   arr[i] = (int*)malloc(8*sizeof(int));
}
for (i=0; i<4; i++)
{
    for(j=0; j<8; j++)
    {
         arr[i][j] = 0;
    }
}
```



Pointer arithmetic ☺

The next code should reset the values of the array.
where is the problem here?

Suppose int size is 4 Bytes.

```
int main()
{
    int x[5] = {1,2,3,4,5};
    int i = 0;
    int *p = x

    while (i < 5) {
         *p = 0;
          p = p + 4;
          printf("%d\n", x[i]);
          i++;
    }
}
```

Pointer arithmetic ☺

The next code should reset the values of the array.
where is the problem here?

Suppose int size is 4 Bytes.

```
int main()
{
    int x[5] = {1,2,3,4,5};
    int i = 0;
    int *p = x

    while (i < 5) {
          *p = 0;
           p = p + 1;
           printf(“%d\n”, x[i]);
           i++;
    }
}
```

Compilation & linkage

Square.h

```
// declaration
int area (int x1, int y1, int x2, int y2);
...
```

Main.c

```
#include "square.h"
int main()
{
  // usage
  area (2,3,5,6);
}
```

Square.c

```
#include “Square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y
2)
{
...
}
```

Preprocessor

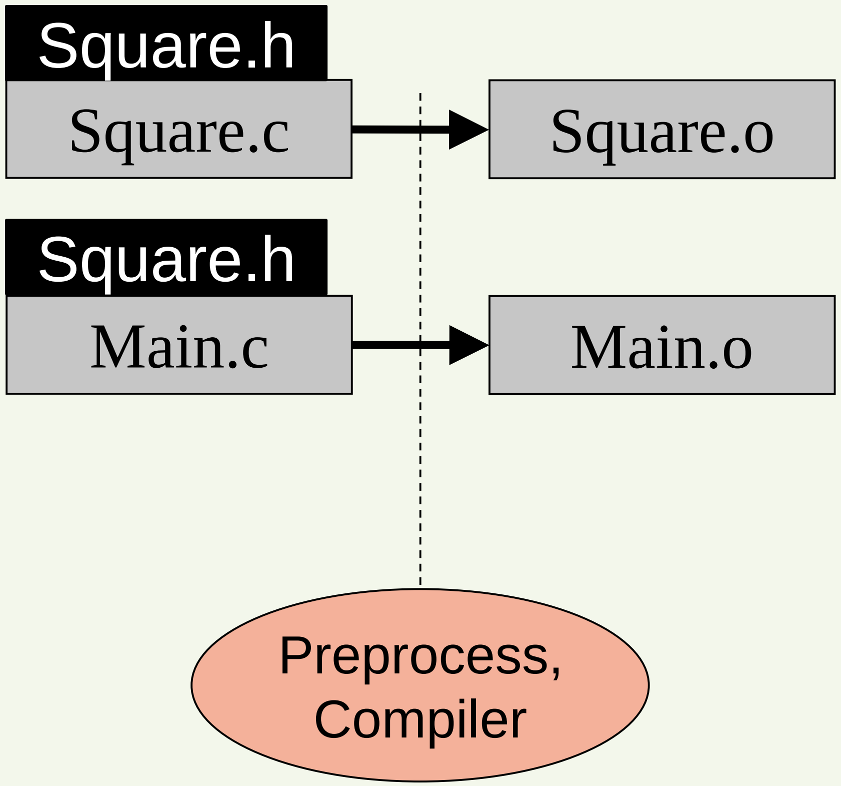
A program which treats #include, #define, and #ifdef commands (+ remove all comments) in .c file.

- #include “square.h” : Copy the content of “square.h” into the c file.
- #define VAR 2: Replace each appearance of VAR to be the integer 2. e.g.: int x = VAR*3;   -> int x = 2*3;
- #ifdef, #ifndef, #else – cut from the code full sections.

When the preprocessor is done there is no #define, #include, #ifdef and programmer comments expressions in the code.

Compiling

- Creates an object file for **each** code file (.c -> .o)
- Each .o file contains opcode of the C code of its *translation unit* (functions, structs, variables etc..)
- Unresolved references still remain

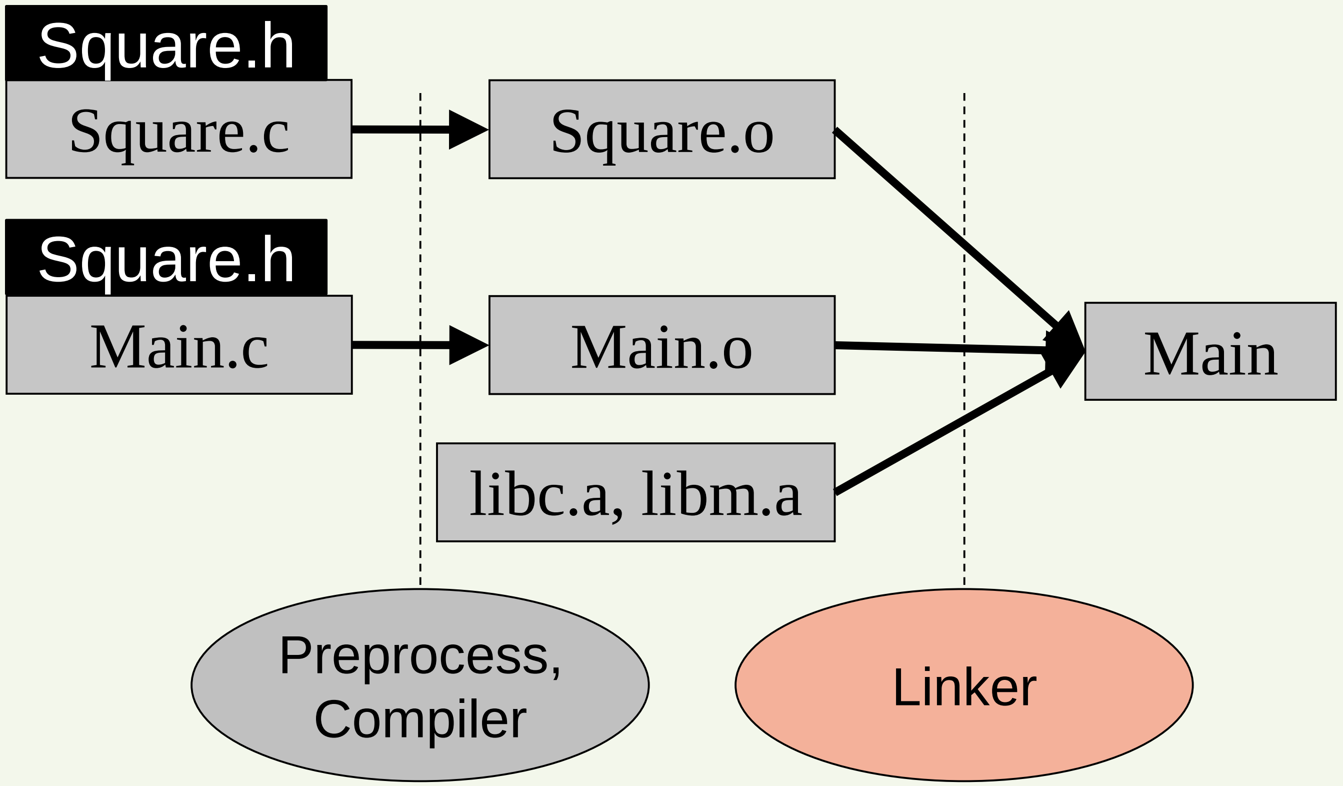


Linking

Combines several object files into an executable file

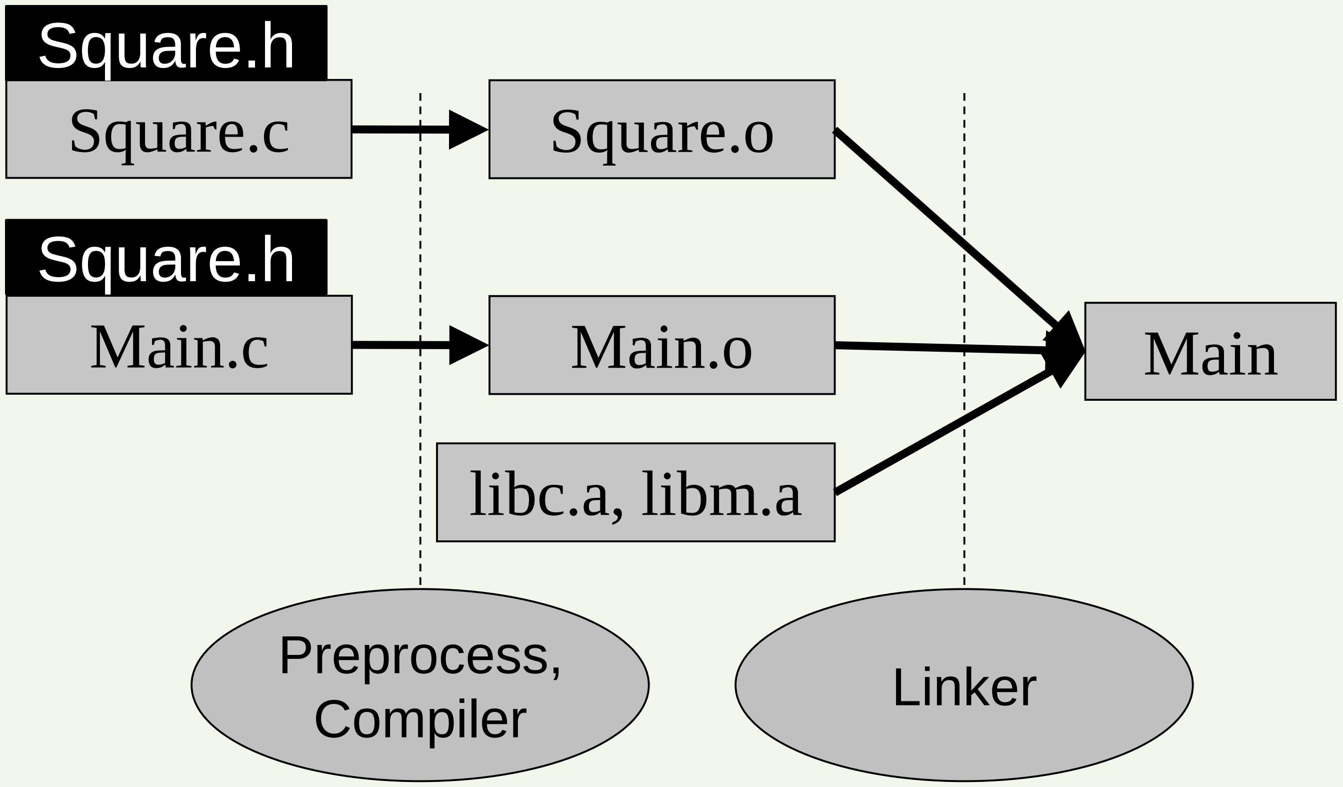
No unresolved references should remain

- Link function calls to function definition code
- Assign symbols to memory addresses



The whole process

```
$ gcc -c -Wall Square.c Square.o
$ gcc -c -Wall Main.c Main.o
$ gcc Square.o Main.o libc.a libm.a -o Main
```



Basic Compilation

- Consider we have to compile the file *driver.c:*
- *gcc -Wextra –c driver.c*
  Creates an object file called driver.o

- *gcc -Wextra driver.o -o driver*
  Creates an executable file called driver.

- Can be done in one line:
- *gcc –Wextra driver.c –o driver*

- **Running the program:**
  Just write the executable name in the command line.

Compilation Errors

- *gcc testFile.c: No such file or directory*
- the problem: wrong name of file, or compiling from the wrong directory

- *testFile.c: In function 'int hello()'*
  testFile.c :12:   syntax error before ';'

Link errors

The following errors appear only at link time

1. Missing implementation

```
> gcc –Wall –o Main Main.c
Main.o(.text+0x2c):Main.c: undefined
reference to `foo'
```

2. Duplicate implementation (in separate modules)

```
> gcc –Wall -o Main Main.o foo.o
foo.o(.text+0x0):foo.c: multiple definition of
`foo'
Main.o(.text+0x38):Main.c: first defined here
```