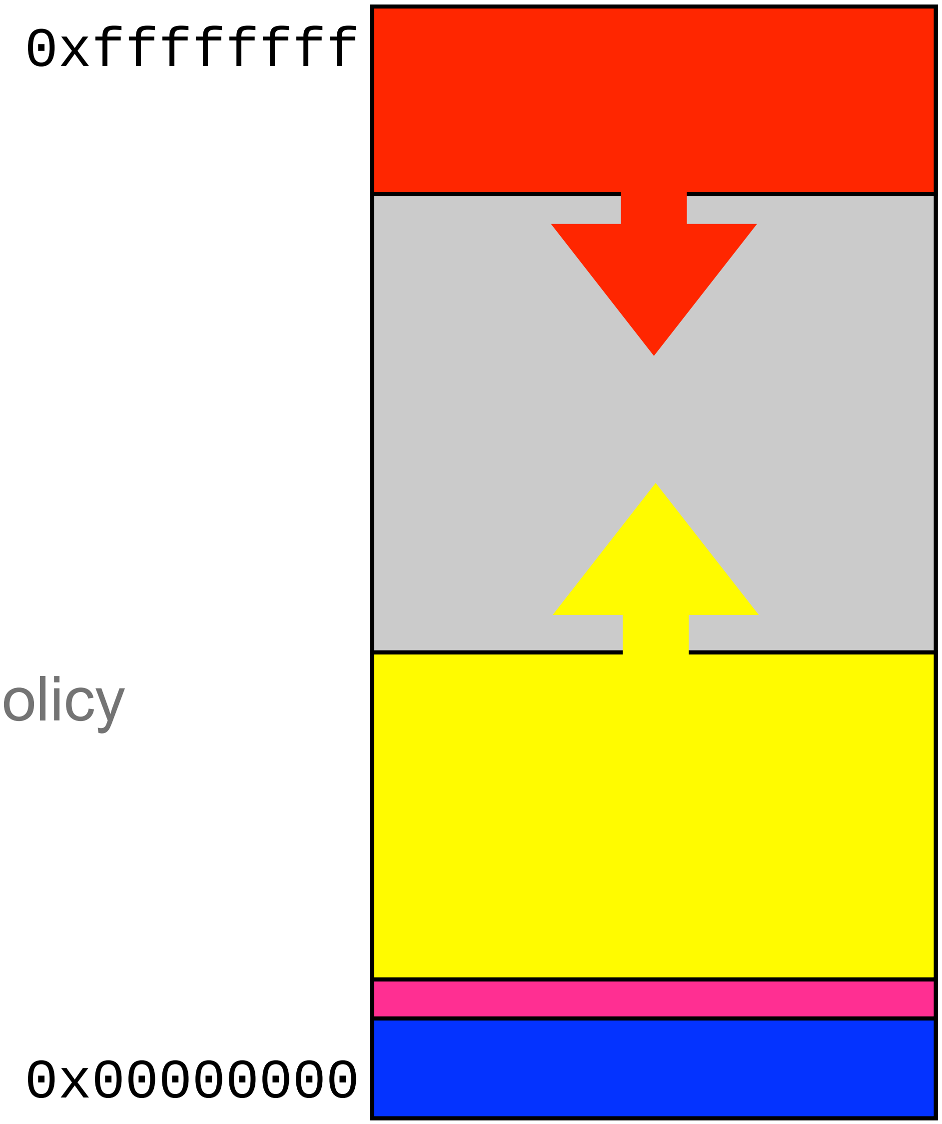
A C Primer (6): Dynamic Memory Allocation

Ion Mandoiu
Laurent Michel
Revised by M. Khan and J. Shi

Recall the memory model...

- Three pools of memory
  - Static/global
  - Stack
  - Heap
- Each pool features
  - Different lifetime
  - Different allocation/deallocation policy



Static/global memory pool

- This is where
  - All constants (including string literals) are held
  - Global variables
  - All variables declared “static” are held
- Allocated when
  - The program starts
- Deallocated when
  - The program terminates
- FIXED SIZE
  - Compiler needs to know the size and make reservations

Stack

- This is where….
  - Memory comes from for local variables in functions!
- Easy to manage because it is automatic!
  - Allocated automatically when entering the function
  - De-allocated automatically when you leave the function
    - Scope is that of function
    - Should not be used after the function returns
      - For example, indirectly used via a pointer

Default stack size using gcc is 2 MBytes.

Need to increase stack size for large arrays and deep recursion

Heap

- This is where…
  - Memory comes from for manual “on-the-fly” allocations
- Who is in charge ?
  - **The programmer** for both allocation / deallocation
- Lifetime of memory blocks ?
  - As long as they are not freed!

Requesting memory on the heap


```
#include <stdlib.h>
void* malloc(size_t size);
```

- size_t is an unsigned integer data type defined in <stdlib.h>
- used to represent sizes of objects in bytes

- **If** successful, a call to malloc(n) returns a generic pointer (void *)
  - It points to a memory block of n bytes on the heap
- If not successful, NULL is returned

```
char* p = malloc(100);  // request 100 bytes
```

Generic pointers: void*

Pointer to a memory block whose content is “un-typed”

- Use for raw memory operations or in generic functions
- Automatic casting when assigned to other pointer types

```
int * pox = malloc(6 * sizeof(int));
```

- Requires casting before dereferencing for read / write

```
*(int *)pv;  // use pv as an int *
```

- NULL, a special pointer value useful for initializations, error handling

```
#define NULL ((void*)0)
```

Can’t always get what you want

- A call to malloc() may fail
  - For example, if you are out of memory
- In this case you get back a NULL value
  - Not much to do except report the error (and terminate nicely)

- Idiom

```
char* p = malloc(100); // request 100 bytes
if (p == NULL) {
   // report error and finish
    perror("Not enough memory");
    exit(1);
}
```

How Much Space ?

- You need to tell malloc() how many **bytes** you need
- To request space for an array, need
  - Number of elements
  - Amount of space for each array element
    - sizeof(T) returns number of bytes needed for a value of type T
- Example

```
int* pox = malloc(6 * sizeof(int)); // request space for 6 ints
if (pox == NULL)
   report error and finish;
```

Another way

```
#include <stdlib.h>

void* calloc(size_t nmemb, size_t size);
```

- calloc() is implemented in terms of malloc()
  - **calloc() also initializes the content to 0**

```
int* pox = calloc(6, sizeof(int)); // request space for 6 ints
if (pox == NULL)
   report error and finish;
```

Adjusting the size

```
#include <stdlib.h>

void* realloc (void* ptr, size_t size);
```

What if you change your mind?

- You requested 100 bytes, but now need 200!

```
char* p = malloc(100); // request 100 bytes
...
p = realloc(p, 200);  // p may change!
```

- Before a call to realloc(p, size), p must be
  - A pointer returned by a previous malloc/calloc/realloc
  - Or NULL, in which case the call is equivalent to malloc(size)

Deallocation

```c
#include <stdlib.h>

void free(void *ptr);
```

- Straightforward
  - Simply call the library function “free”
  - Takes a pointer to the block to free

```c
free(ptr);
```

- Do not free a pointer twice!
  - After freeing a pointer, set it to NULL!

Two key rules

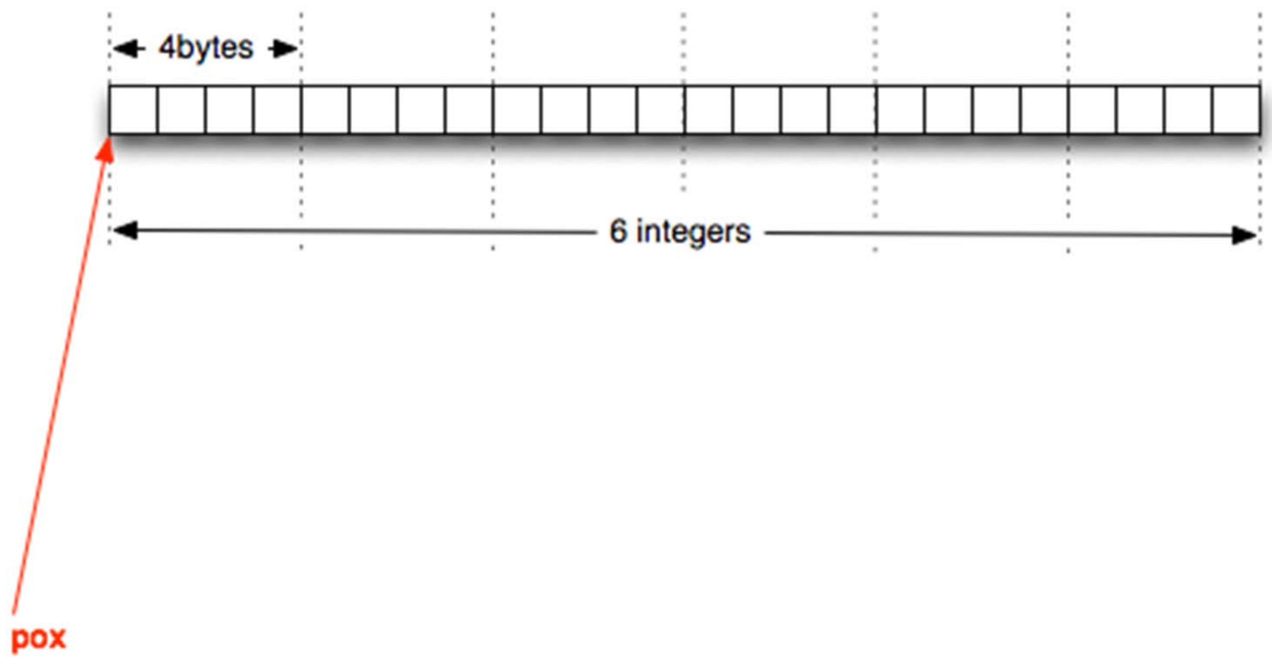
Rule 1: Everything you requested should be freed, eventually

Rule 2: Only free what is allocated via malloc/calloc/realloc

- Consequences of not following the rules
  - **Memory “leaks”**
    - Your program will eventually run out of memory
  - **Undefined behavior and horrible crashes**
    - Freeing unallocated memory or already freed memory
      - May cause a memory error and a program crash
      - Worse, may corrupt the heap and cause a crash later
      - Even worse, the program may keep running, totally corrupting your data, and writing it to disk without you realising

Pointers and arrays

`After pox = malloc(6 * sizeof(int));`



- That looks like an array!!!
  - And you **can** use pox as if it is an array

Example: use pointer as array

```
// programmers have to manage memory themselves before C99
void doSomething(int n) {
    int * pox;
    pox = malloc(sizeof(int)*n); // request mem from heap
    *pox = 0;    // set the int at the address pox to 0
    pox[0] = 0;  // same thing
    pox[1] = 1;  // the int after pox[0]
    // more lines here ...
    free(pox);   // remember to free!
}
```

Array of pointers

- Pointers, like integers, can be placed in an array

```
int  a0;
int a1;
int a2;


char *p0 = malloc(10);
char *p1 = malloc(10);
char *p2 = malloc(10);
```

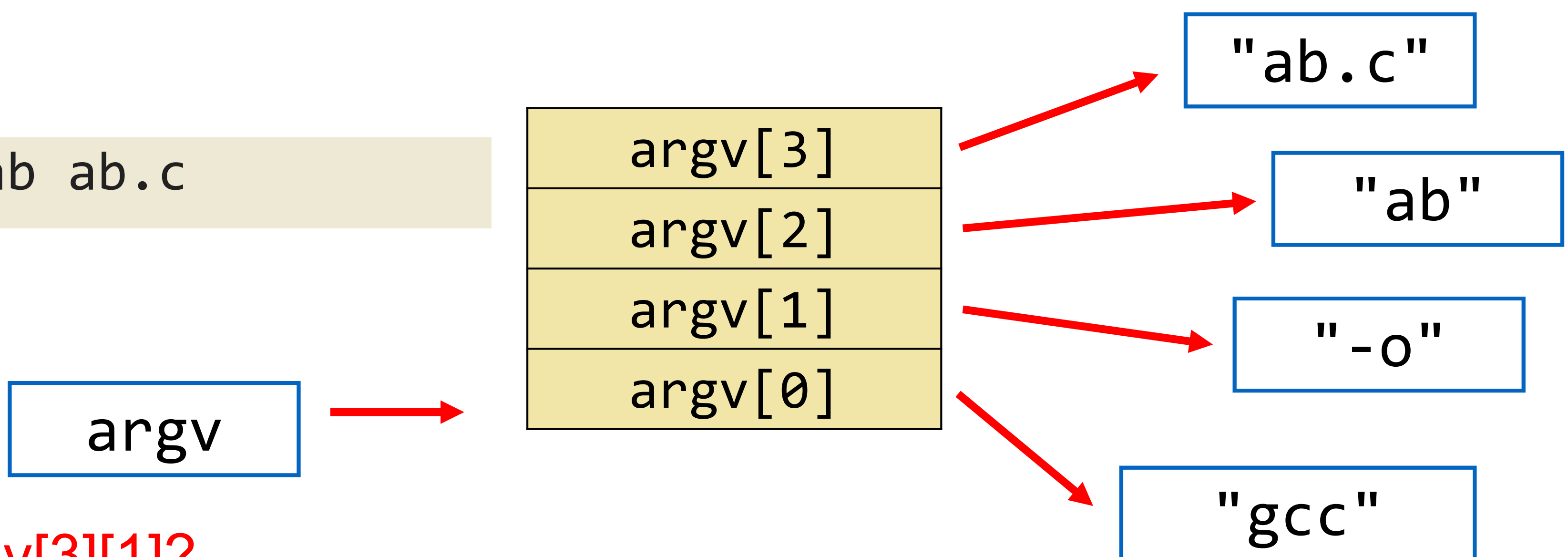
```
// array of int's
int  a[3];

// array of pointers
char * p[3];

// Can also do with a loop
p[0] = malloc(10);
p[1] = malloc(10);
p[2] = malloc(10);
```

Example: command line arguments

```
int main (int argc, char *argv[]);
```

- argc: the number of arguments on the command line
- argv: array of pointers to characters
  - The number of elements is argc
  - Each element in an array points to null-terminated strings
- Example:

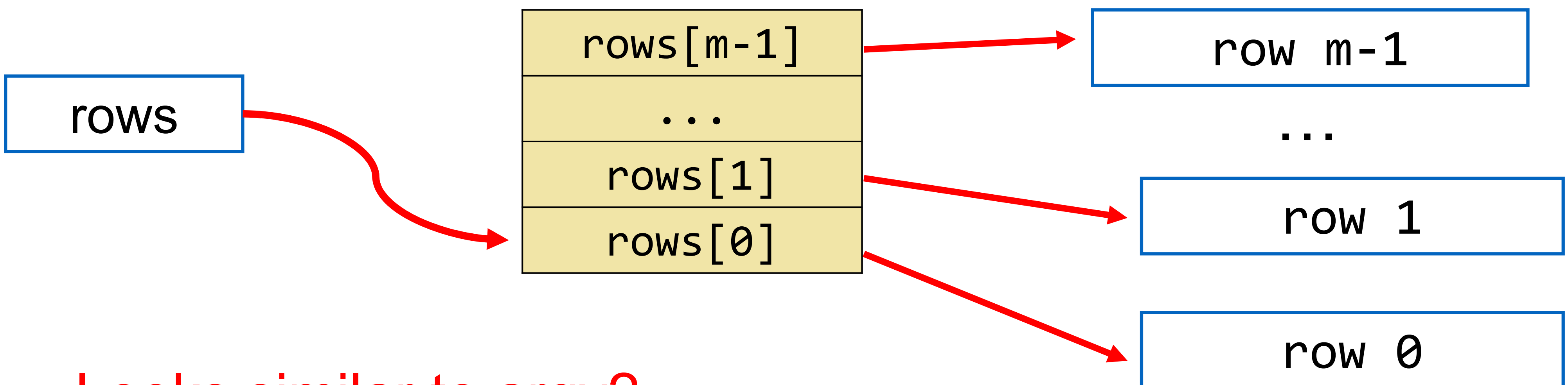
```
$gcc -o ab ab.c
```



What is argv[3][1]?

Example: allocating 2d dynamical array

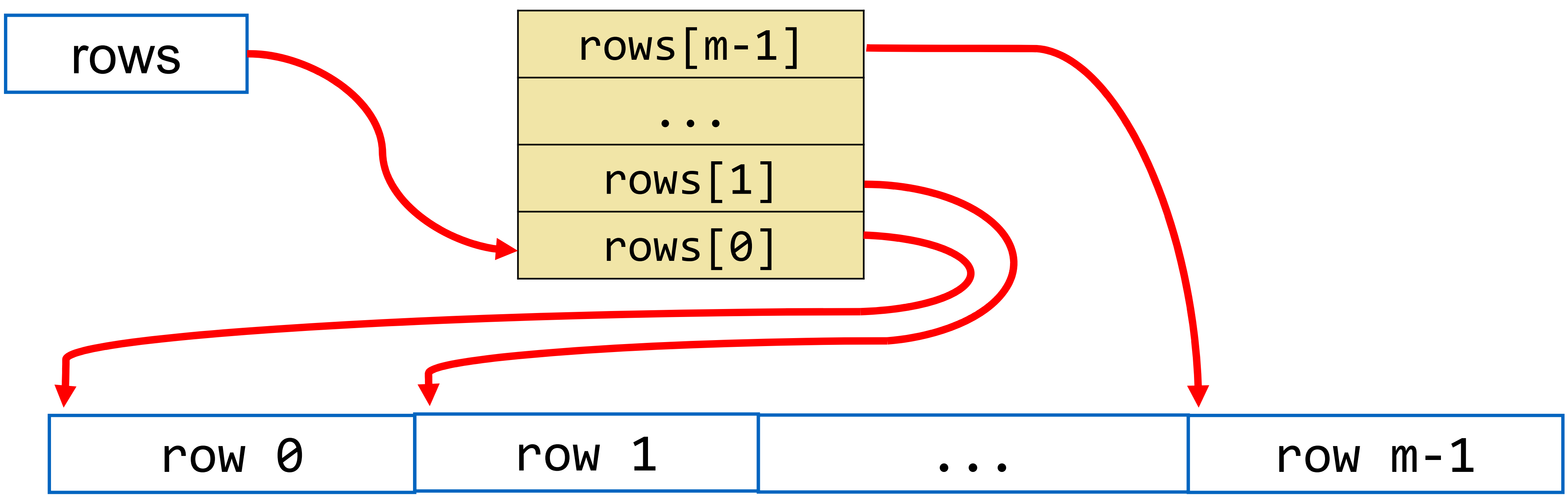
```
void doSomething(int m, int n)
{ int **rows;
  rows = malloc(sizeof(int *) * m);   // array of pointers
  for (int i = 0; i < m; i++)
    rows[i] = malloc(sizeof(int)*n); // one int array for each row
  ...
  for (int i = 0; i < m; i++)
    free(rows[i]);
  free(rows);
}
```



Looks similar to argv?

Example: allocating 2d dynamical array (take 2)

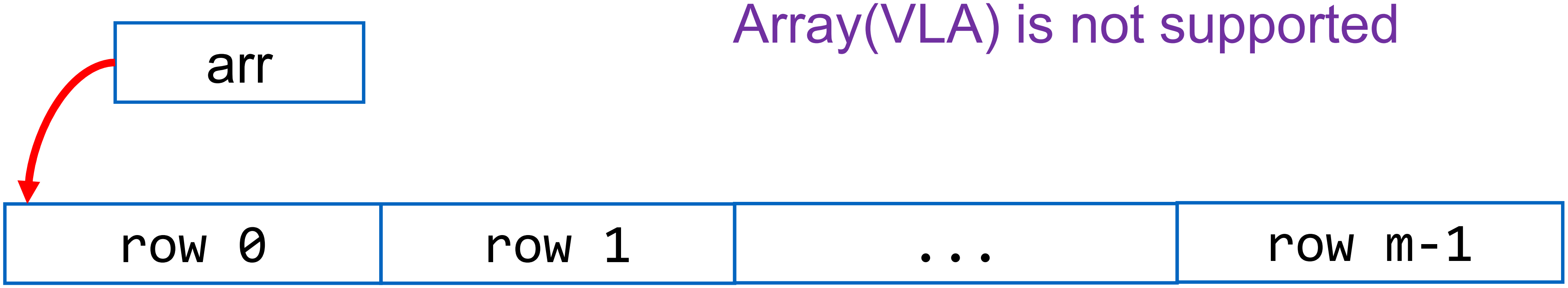
- Request all the memory space needed by data with one malloc() call
  - Instead of call malloc() m times, for each row
- Calculate rows[1], rows[2], and so on (pointer arithmetic! next lecture)
  - You can even calculate the address of each element (e.g. rows[10][5])



Example: allocating 2d dynamical array (take 3)

```
void doSomething(int m, int n)
{ int (* arr)[n];     // arr is a pointer to an array of n int's
  arr = malloc(m * n * sizeof(int)); // one malloc() for data
  // to access elements
  arr[1][2] = 1; arr[2][3] = arr[1][2] + 10;
  ...
  free(arr); // free memory
}
```

n must be constant if Variable Length Array(VLA) is not supported



Study the remaining slides yourself

Pointers taking the address of …

- A static ?
  - The address is never going to go “bad”
  - The static lives as long as the program!
- A stack [automatic] variable ?
  - The address is valid as long as the variable is!
  - When the function returns…. The address is bogus
- A heap variable ?
  - The address is valid as long as the variable is!
  - The variable disappears when explicitly de-allocated (freed)