CS 162
Intro to Computer Science II

Lecture 5

Memory Model

Dynamic arrays

1/24/24

Oregon State University

Pointer Summary

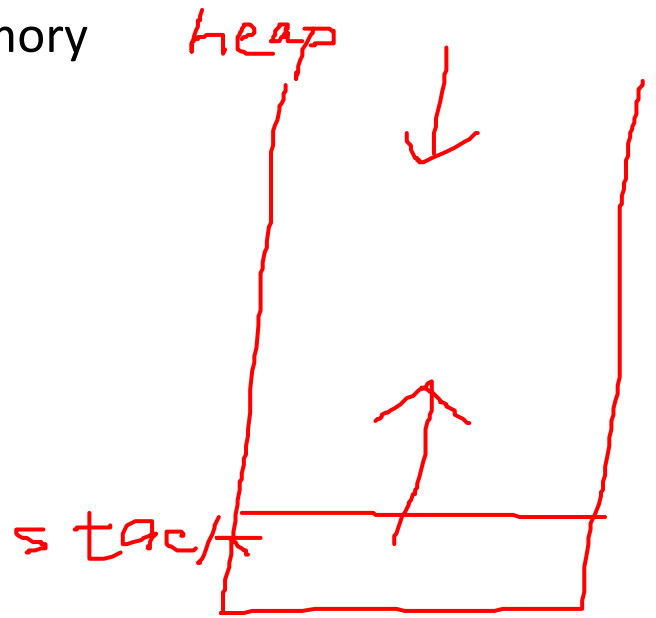
- To summarize:
  - We can declare pointer variables to store addresses (not data) using the syntax T* where T is some type (e.g. int *p)
  - We can get the address of some variable using the & operator (e.g. &x, &y)
    - Most often, this would then be assigned to a pointer variable (e.g. p = &x)
  - We can dereference a pointer (i.e. follow a pointer) to get the data from the address it stores by using the * operator (e.g. cout << *p << endl)
  - We can change the address the pointer stores to have it reference some other variable (e.g. p = &z)

Lecture Topics:

- Memory Model
- Dynamic array

Program Memory

- In a C++ program, there are two distinct areas of memory in which we can store data, the stack and the heap.
  - **Stack** – a limited-size chunk of the larger blob of system memory
  - **Heap** – comprises essentially all the rest of system memory



- The stack and the heap grows towards each other

Stack

- Stack is small (general 8 MB)
  - If running out of stack memory → program crash (stack overflow)
- Stack memory is allocated in contiguous block during compile time
  - Known as static memory
- Stores global/local variables, constants, and values declared in a program's functions
- Functions have their own stack frame
- When a function is called (in use), it is pushed onto the stack
- When a function ends, the stack frame collapses and cleans/frees up the memory for you (automatically)

Heap

- Heap is larger (determined by the size of RAM)
- Heap memory is allocated in random order during run time
  - Known as dynamic memory
- Allocated with pointers and the new operator, i.e.,
  - `int *p = new int; //new returns an address on the heap`
- Dynamic memory does not disappear when the function ends as they are on the heap and not the function stack
- Can run out of heap space → heap overflow!
- Must manually free (delete) heap memory after used, otherwise memory leaks
  - `delete p;`

Demo: Stack vs. Heap Memory

Lecture Topics:

- Dynamic array

Dynamic Array Motivation

- Q1: We want to allocate an array of integers, but I don't know the size until the user inputs it. What size should I use when declaring my array?
  - `int numbers[??];`
  - Note: `int numbers [var]` is not supported by all C/C++ compilers and **considered bad practice**!

- Q2: What if we need that array to KEEP ALIVE after our function ends?

- Both questions are solved with dynamic memory (aka. runtime memory)

1D Dynamic Array

- Creation:
  - `int *arr = new int [5];`
- Deletion:
  - `delete [] arr; //check memory leaks using valgrind`
- Passing 1D dynamic array into function:
  - Same as 1D static array, i.e., pass the pointer

```
void pass_1darray(int *a) {…}
```

OR

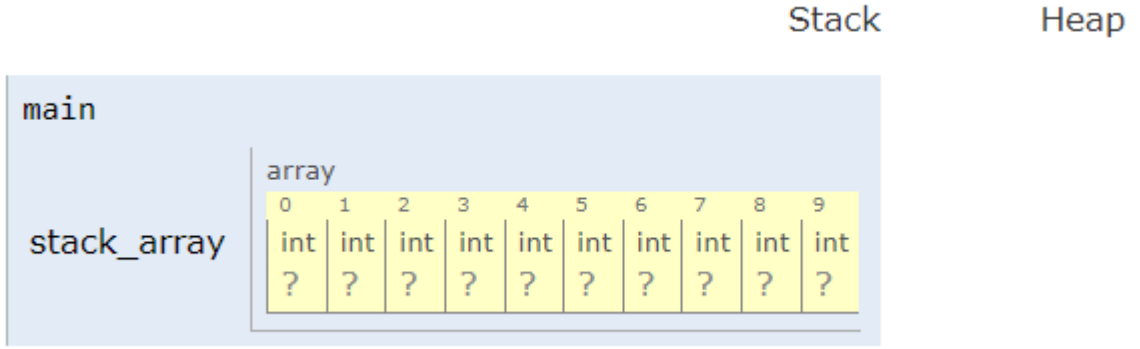
```
void pass_1darray(int a[]) {…}
```

  - Function call: `pass_1darray(arr);`

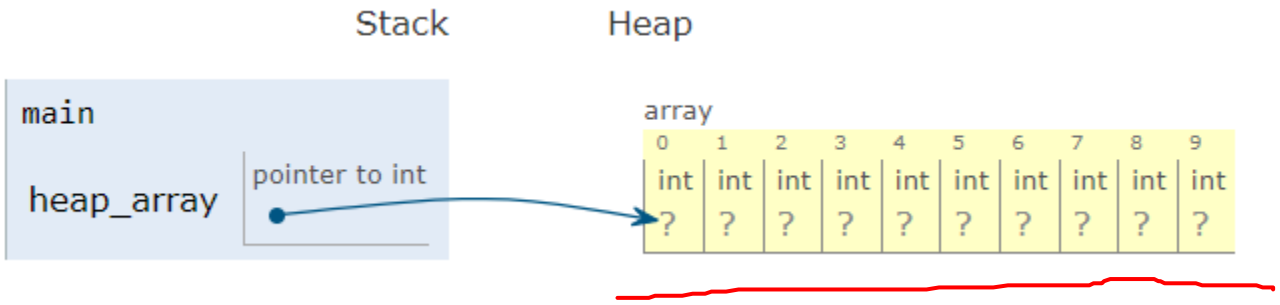
- Demo…

Static vs. Dynamic 1-D arrays…

```
1  int main() {
2    int stack_array[10];
3
4    return 0;
5  }
```




```
1  int main() {
2    int *heap_array = new int [10];
3
4    return 0;
5  }
```

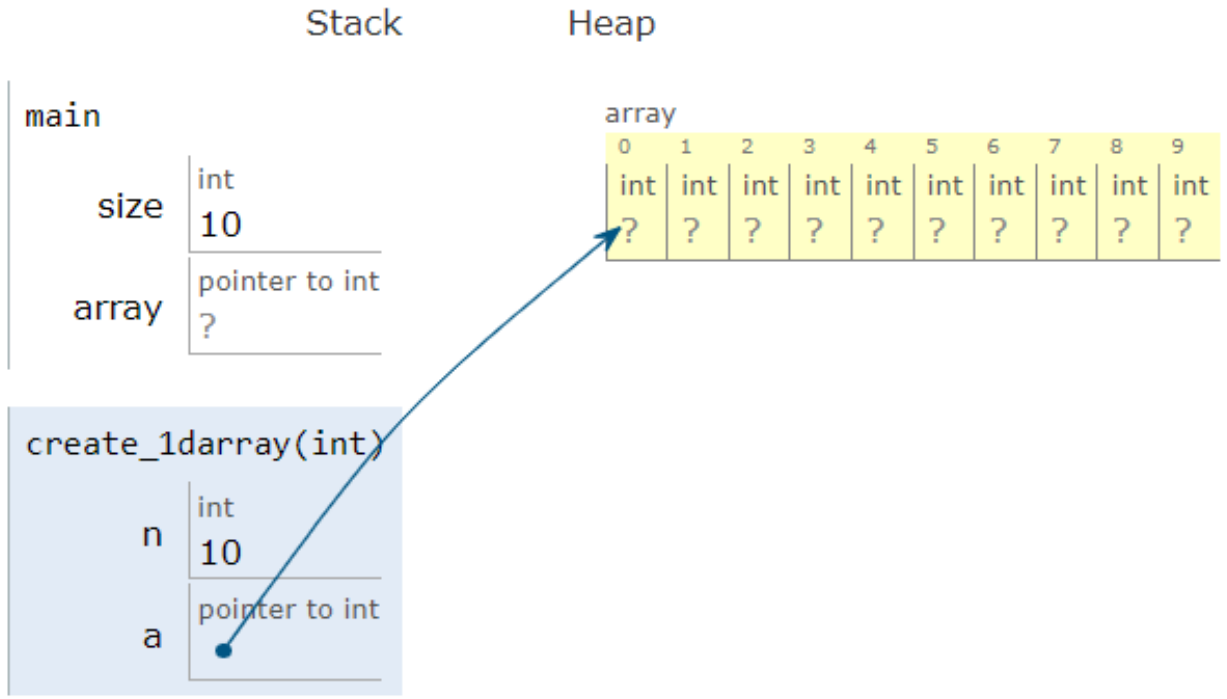


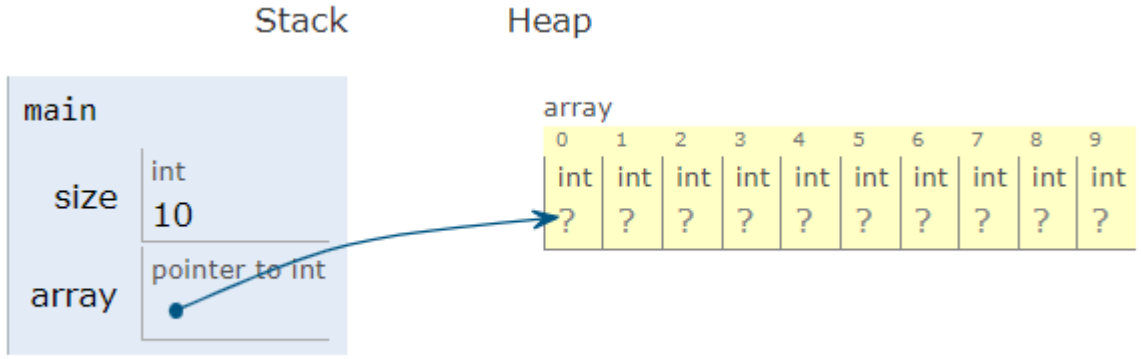
Exercise

- How do I initialize an int array in a function?
- How can I print the contents of the int array in a function?
- How would I create a dynamic int array using a function? (3 ways)
  - int* create_array1(int size);
  - void create_array2(int *&array, int size);
  - void create_array3(int ** array, int size);

Create 1-D Array in Functions

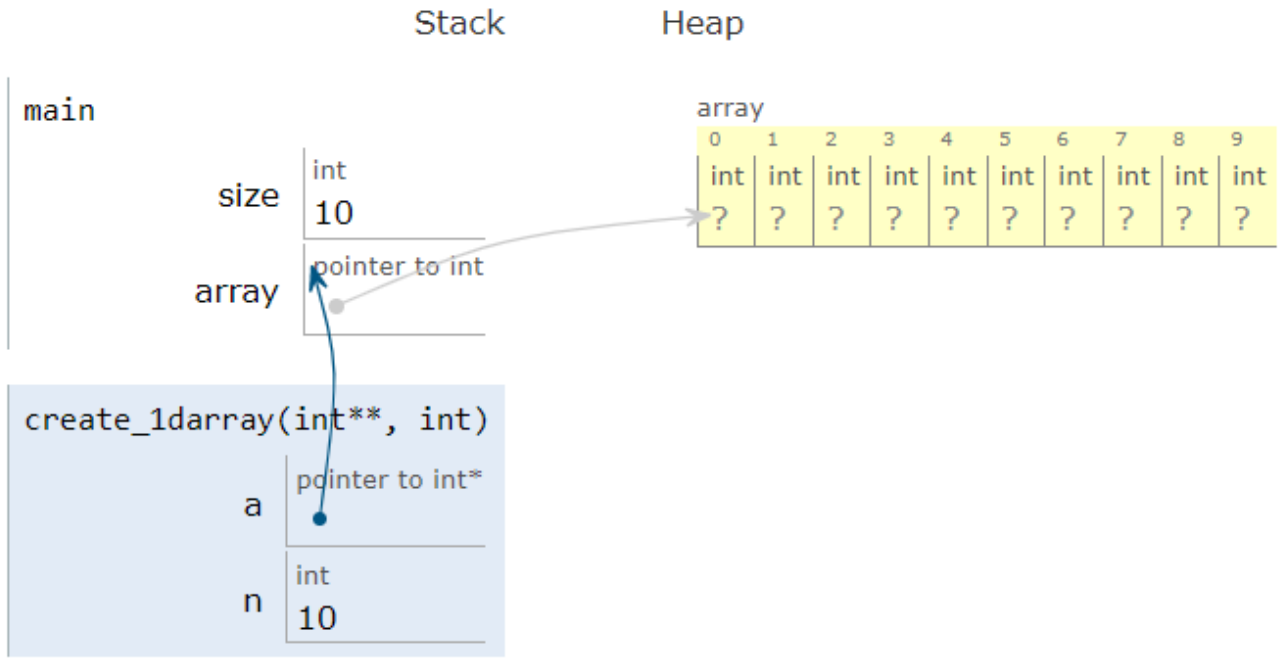
```
int main() {
    int *array;
    …
    array = create_1darray(size);
    …
}
int *create_1darray(int n) {
     int *a = new int [n];
     return a;
   }
```

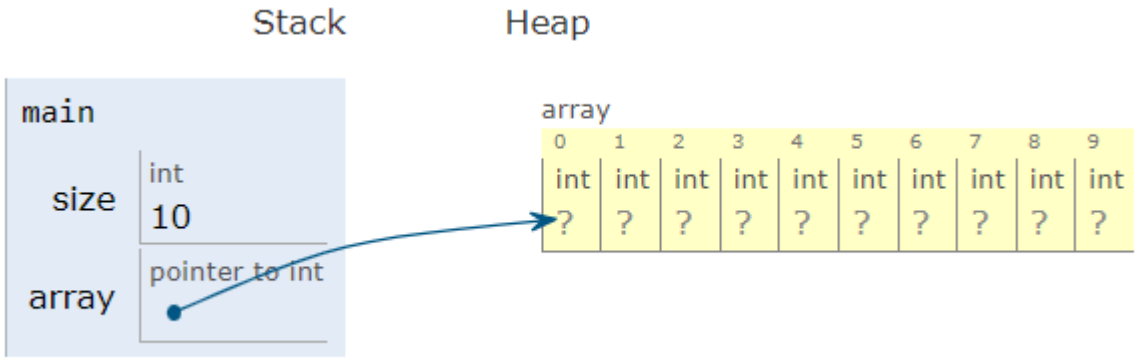




Create 1-D Array in Functions

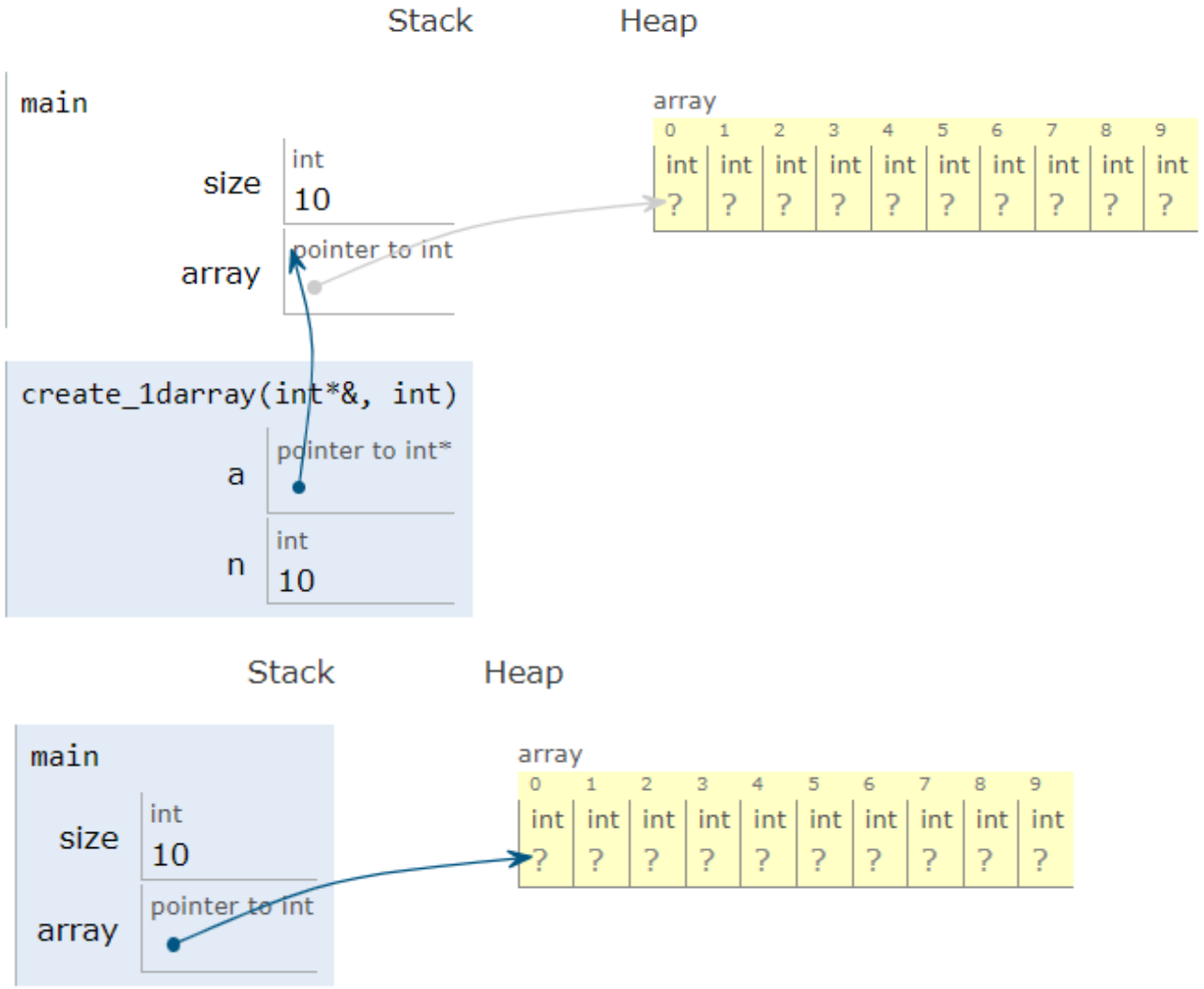
```
int main() {
   int *array;
   …
create_1darray(&array, size);
   …
}
void create_1darray(int **a, int n) {
    *a = new int [n];
}
```





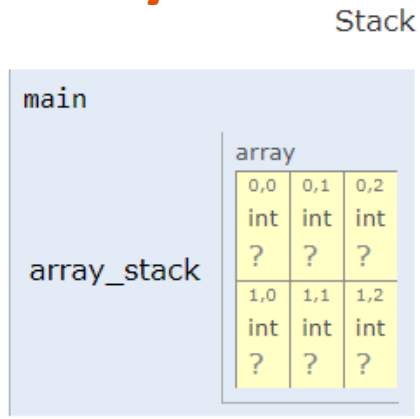
Create 1-D Array in Functions

```
int main() {
   int *array;
   …
create_1darray(array, size);
   …
}
void create_1darray(int *&a, int n) {
    a = new int [n];
}
```

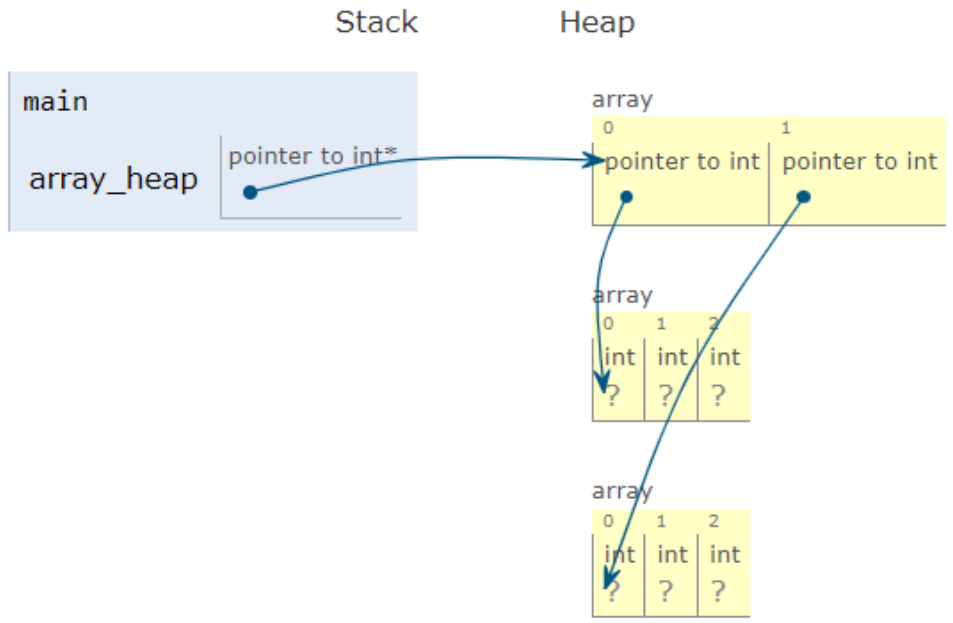


Static vs. Dynamic 2-D arrays…

```
1  int main() {
2    int array_stack[2][3];
3
4    return 0;
5  }
```

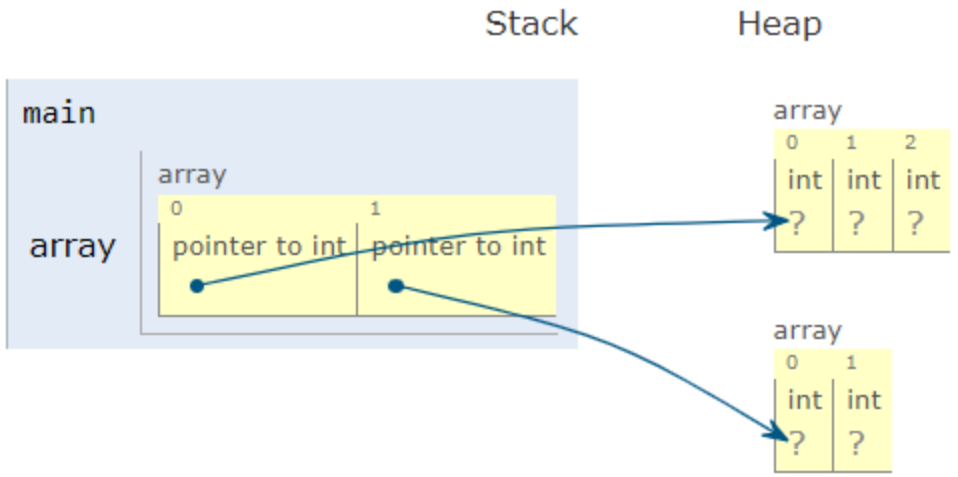


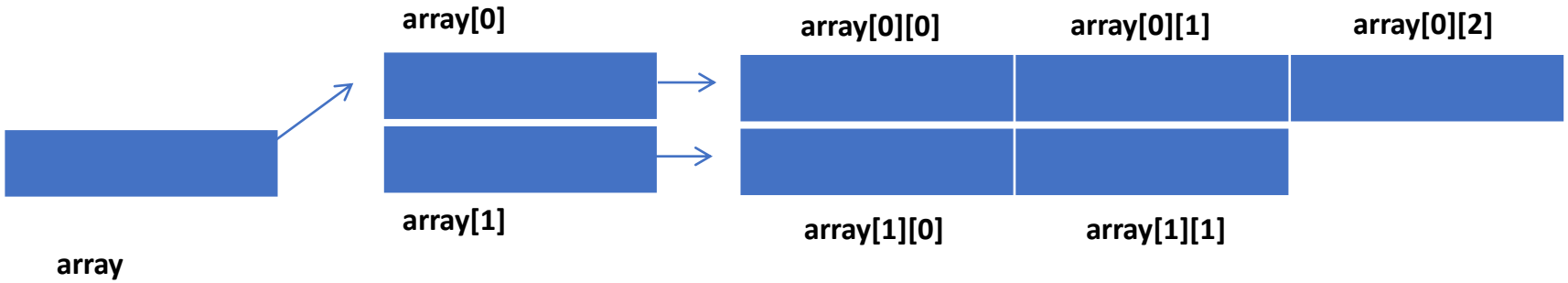

```
1  int main() {
2    int **array_heap = new int* [2];
3    for(int i = 0; i < 2; i++)
4      array_heap[i] = new int [3];
5
6    return 0;
7  }
```



Jagged Arrays

```
int *array[2];
array[0] = new int[3];
array[1] = new int[2];
```





Passing a 2-D Array (Dynamic)

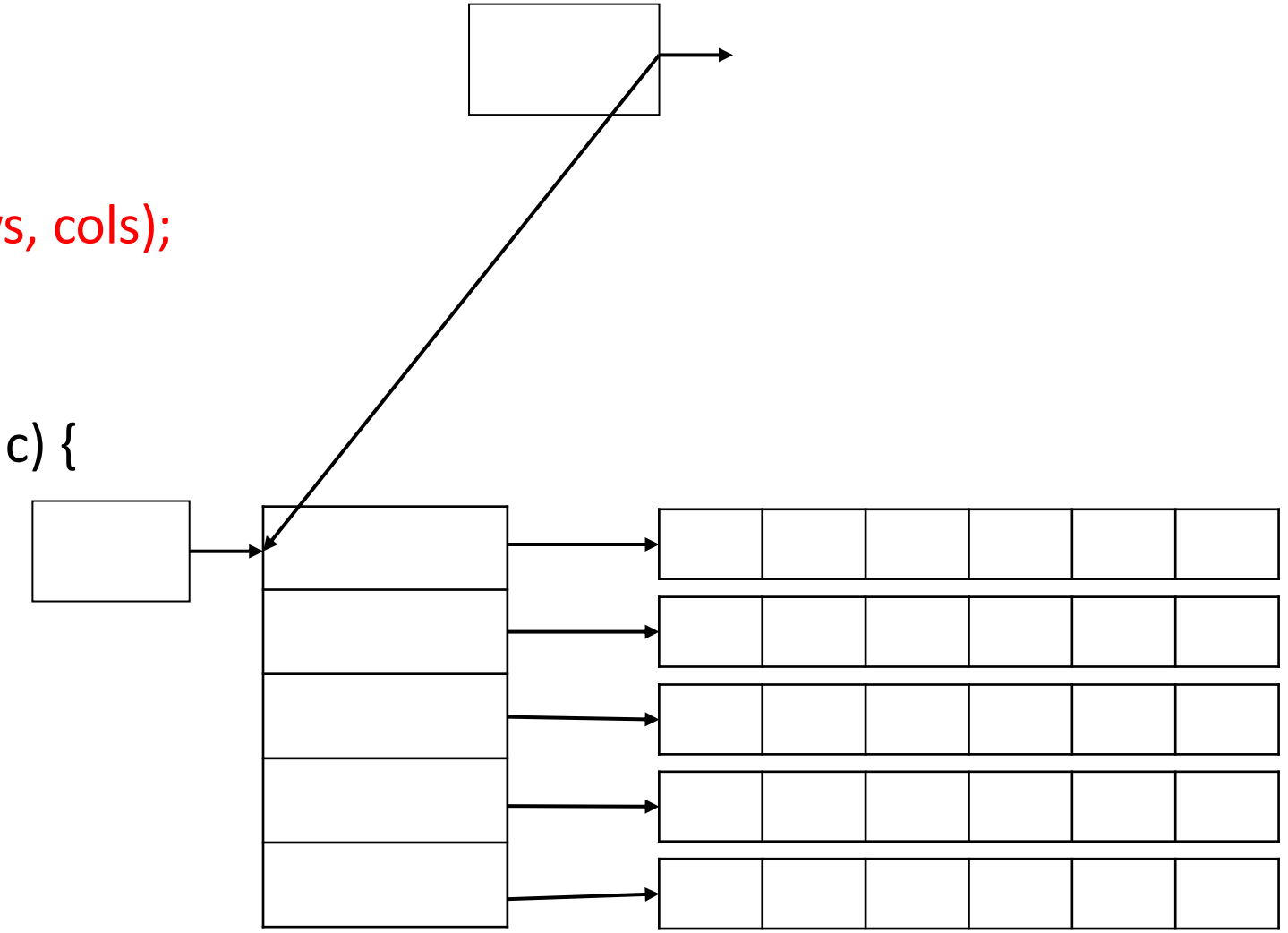
```
int main() {
    int **array;
    …
    pass_2darray(array, row, col);
    …
}
void pass_2darray(int *a[], int row, int col) {
     cout << “Array at zero: ” << a[0][0] << endl;
}
```

OR

```
void pass_2darray(int **a, int row, int col) {
    cout << “Array at zero: ” << a[0][0] << endl;
}
```

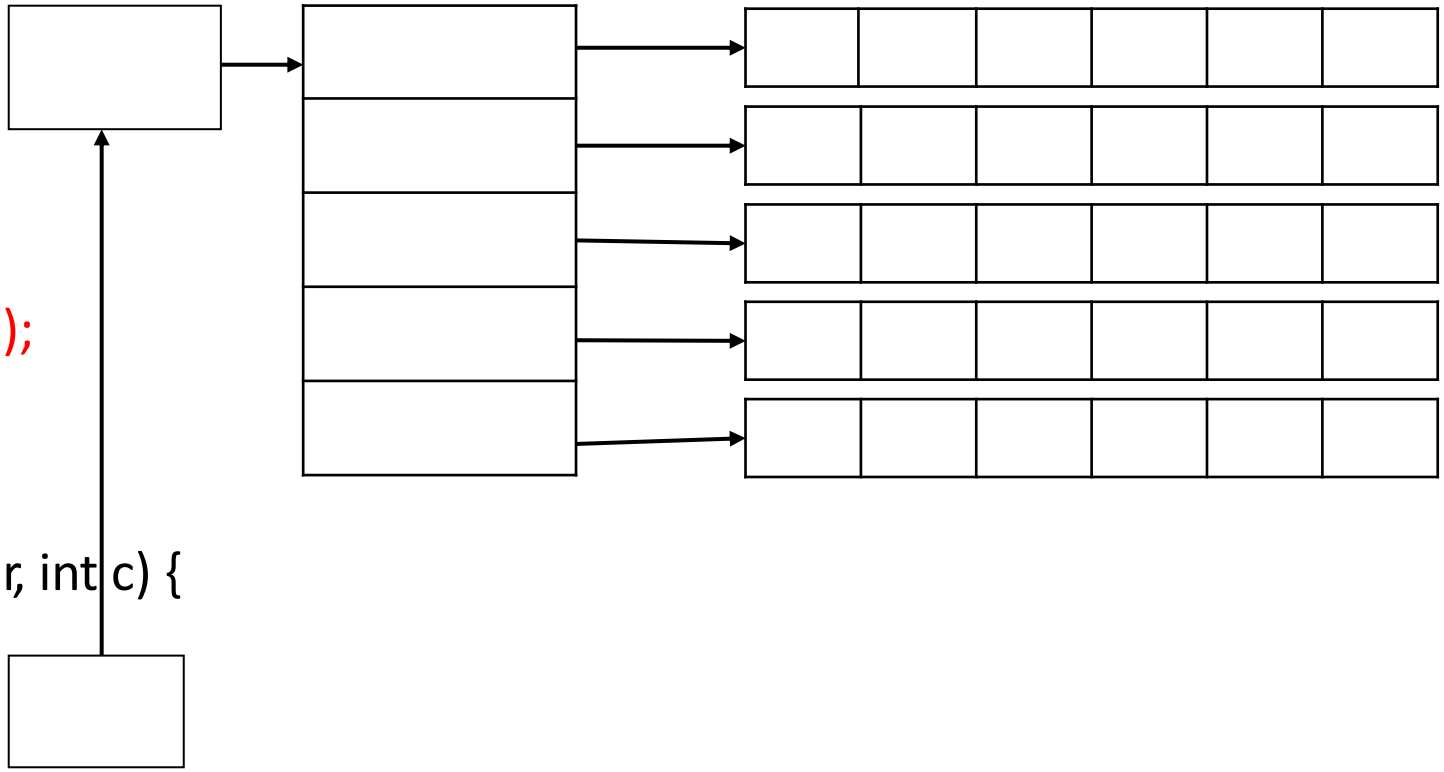
Create 2-D Array in Functions

```
int main() {
    int **array;
    …
    array = create_2darray(rows, cols);
    …
}
int **create_2darray(int r, int c) {
     int **a;
     a = new int*[r];
     for(int i=0; i<r; i++)
        a[i] = new int[c];
     return a;
   }
```



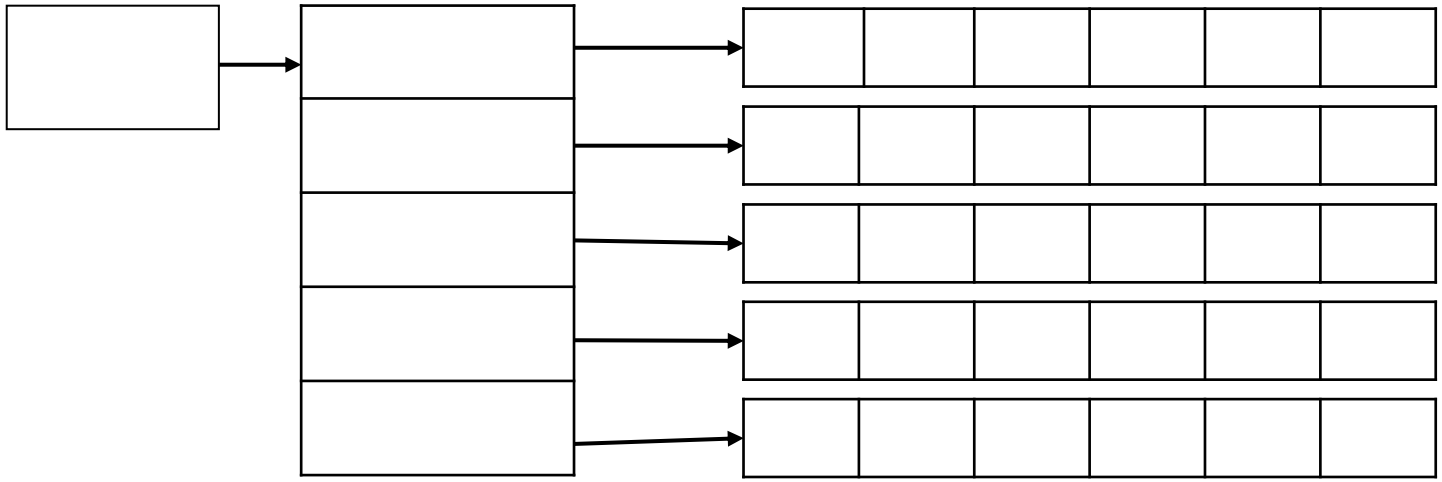
Create 2-D Array in Functions

```
int main() {
   int **array;
   …
create_2darray(&array, rows, cols);
   …
}
void create_2darray(int ***a, int r, int c) {
    *a = new int*[r];
    for(int i=0; i<r; i++)
        (*a)[i] = new int[c];
}
```



Create 2-D Array in Functions

```
int main() {
   int **array;
   …
create_2darray(array, rows, cols);
   …
}
void create_2darray(int **&a, int r, int c) {
    a = new int*[r];
    for(int i=0; i<r; i++)
        a[i] = new int[c];
}
```

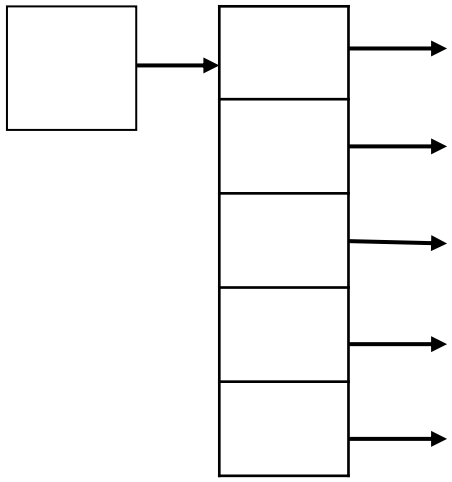


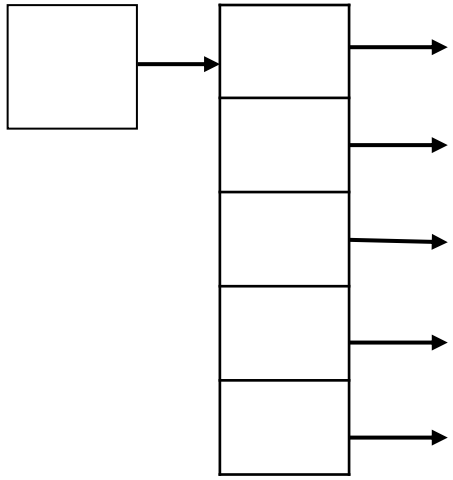
How does freeing memory work in 2D arrays?

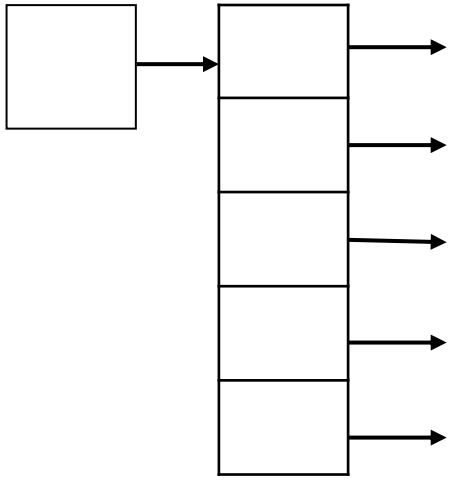
```
int *r[5], **s;

for(int i=0; i < 5; i++)
    r[i]=new int;
for(int i=0; i < 5; i++)
    delete r[i];
```

```
for(int i=0; i < 5; i++)
    r[i]=new int[5];
for(int i=0; i < 5; i++)
    delete [] r[i];
```

```
s=new int*[5];
for(int i=0; i < 5;
i++)
    s[i]=new int[5];
for(int i=0; i < 5;
i++)
    delete [] s[i];
delete [] s;
```







Lecture Topics:

- Structs

Structures

- Data Structures so far…
  - Variables
  - Arrays

- What if we want mixed types?
  - Record: name, age, weight, etc. of a person
  - Use **struct** type

Structs

- User defined composite data type
- Container which holds many variables of different types
- Can be used as any other data type with some extra features
- The instances created by such data type are called objects (items)

How to define a struct?

```
// definition of a Book struct
struct Book {
        int pages;
        string title; // a string inside the struct
        int num_authors;
        string* authors; // a pointer to a string
};

// declare a Book object (item)
Book text_book;

// declare and initialize at the same time
Book b1 = {.pages = 150, .title = "Harry Potter", .num_authors = 2};
//or
Book b1 = {150, "Harry Potter", 2};
```

Note: in order, non-skip

Working with structs

- Can use the same way as any other type
- The dot operator(.) allows us to access the member variables

```
Book bookshelf[10];
for (int i = 0; i < 10; ++i){
     bookshelf[i].num_pages = 100;
     bookshelf[i].title = “Harry Potter”;
     bookshelf[i].num_authors = 2;
     bookshelf[i].authors = new string[2];
}
```

Using pointers with structs

```
Book bk1; //statically allocated
Book* bk_ptr = &bk1;

//dereference the pointer and access the data member
(*bk_ptr).title = "Harry Potter";

//a shortcut to dereference the pointer to the struct
// the arrow (->) operator
bk_ptr -> title = "The Cars";
bk_ptr -> num_pages = 259;

//this works for objects on the heap as well
Book* bk_ptr2 = new Book;
bk_ptr2 -> title = "Transformers";
```

Demo