CS 162
Intro to Computer Science II

Lecture 4

Pointers

Memory Model

1/22/24

Oregon State University

Odds and Ends

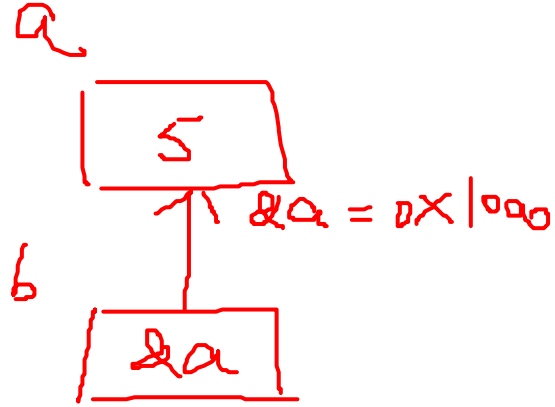
- Design 1 past due, expected grades back by this Friday

Lecture Topics:

- Pointers
  - Pointers vs. references
- Memory Model
- Dynamic Arrays

C/C++ Pointers

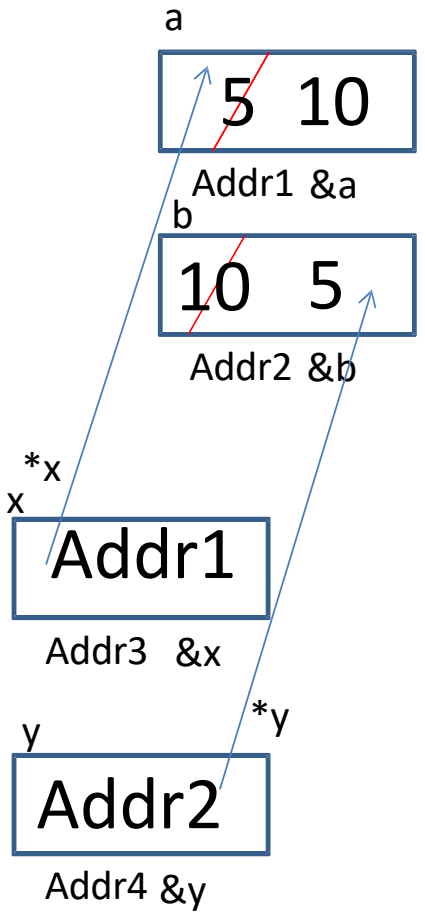
1. value / content
2. name
3. type
4. addr.



- Pointers == variables that hold memory addresses
- Variable declaration: `int a = 5;`
  - Creates a variable on the stack of size int with the value 5
- Pointer declaration: `int *b = &a;`
  - Creates a pointer variable on the stack which can hold an address of an int and sets the value of the pointer (the address the pointer points to) to the address of `a`
- Dereferencing Pointer: `cout << *b << endl;`
  - Dereference: access the value stored in the memory address held by a pointer
  - Will print the value stored at the address which `b` points to
- Every pointer points data of a specific data type

C++ Pointers

```
void swap(int *, int *);
int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    cout << "a: " << a << "b: " << b;
}
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```



Pointer and References Cheat Sheet

- &
  - If used **in a declaration** (which includes function parameters), it **creates and initializes** the reference.
    - Ex. void fun (int &p); //p will refer to an argument that is an int by implicitly using *p (dereference) for p
    - Ex. int &p=a; //p will refer to an int, a, by implicitly using *p for p
  - If used **outside a declaration**, it means **"address of"**
    - Ex. ptr=&a; //**fetches the address of** a (only used as rvalue!!!) and store the address in ptr. (ptr is a pointer variable)

Pointer and References Cheat Sheet

- *
  - If used **in a declaration** (which includes function parameters), it **creates** the pointer.
    - Ex. int *p; //p will hold an address to where an int is stored
  - If used **outside a declaration**, it **dereferences** the pointer
    - Ex. *p = 3; //**goes to the address** stored in p and stores a value
    - Ex. cout << *p; //**goes to the address** stored in p and fetches the value

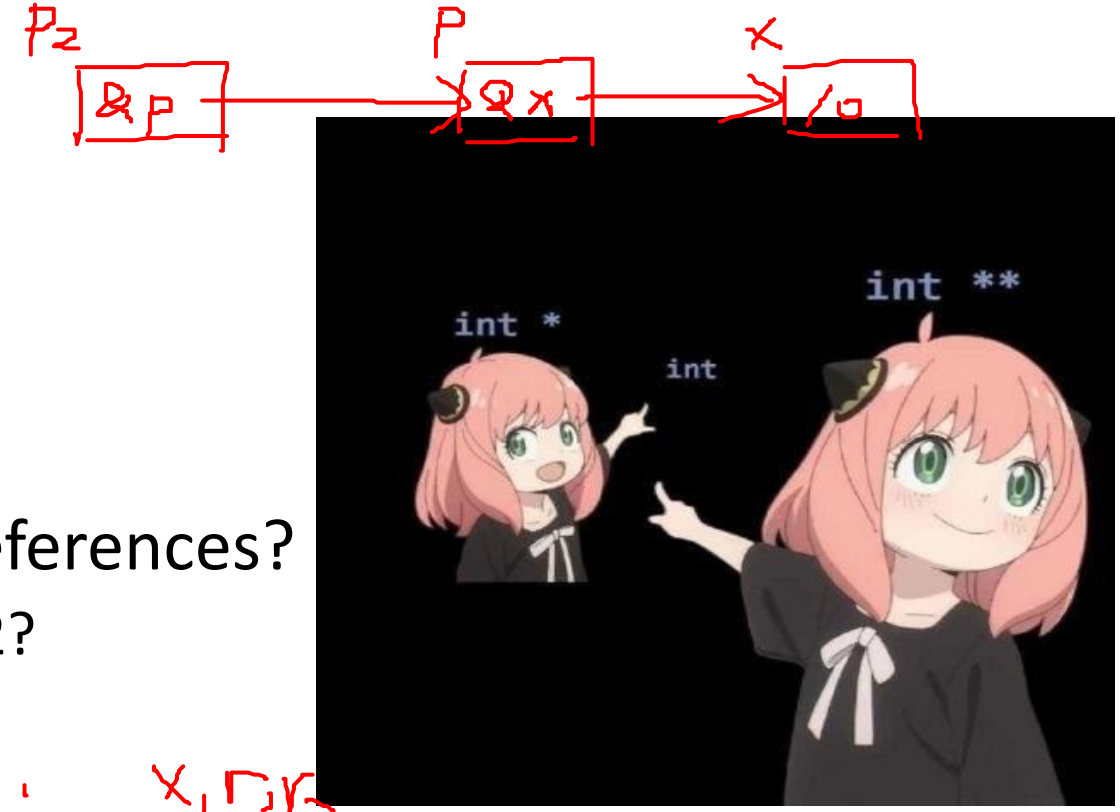
- Check point: How to separate the following into two statements?

```
int *p = &a; //declare an int pointer and initialize it to &a
```

Exercise: Pointers vs. References

int x = 10;
int *p = &x;
int **p2 = &p;

- What if you made a pointer (p2) that points to a pointer (p) that points to an int (x)?
  - What would the picture look like?
  - Write the code for this picture.



- Can you make this same picture for references?
  - What if you had two references, r and r2?

int &r = x;

int &&r2 = r; ?

int &r2 = r;

x, r, r2

10

& and * Summary

- &<variable> evaluates to the “address-of” <variable>

- *<pointer> dereference the <pointer>
  - (data at the address given by <pointer>)

- & and * are inverse operations
  - &value → address
  - *address → value
  - *(&value) → value

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:

- Pointers (cont.)
  - Pointer vs. Reference
- Memory Model
- Dynamic Arrays

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

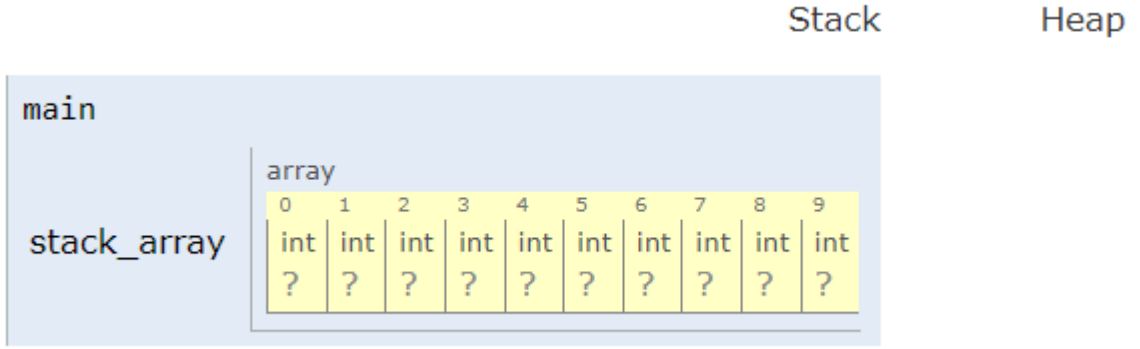
```
void pass_1darray(int a[]) {…}
```

  - Function call: `pass_1darray(arr);`

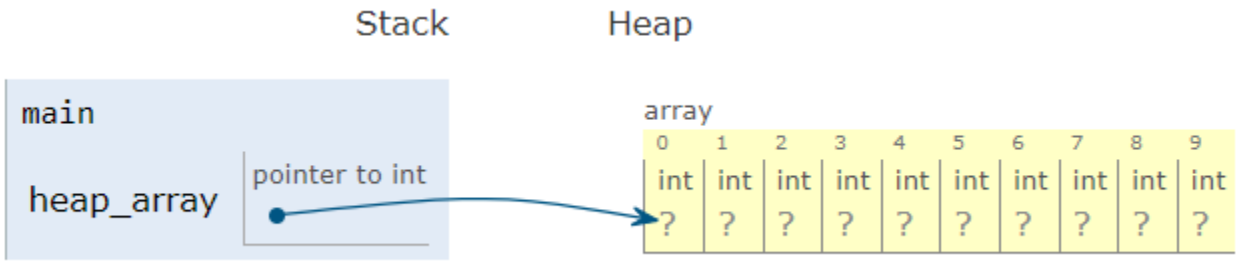
- Demo…

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  }
```

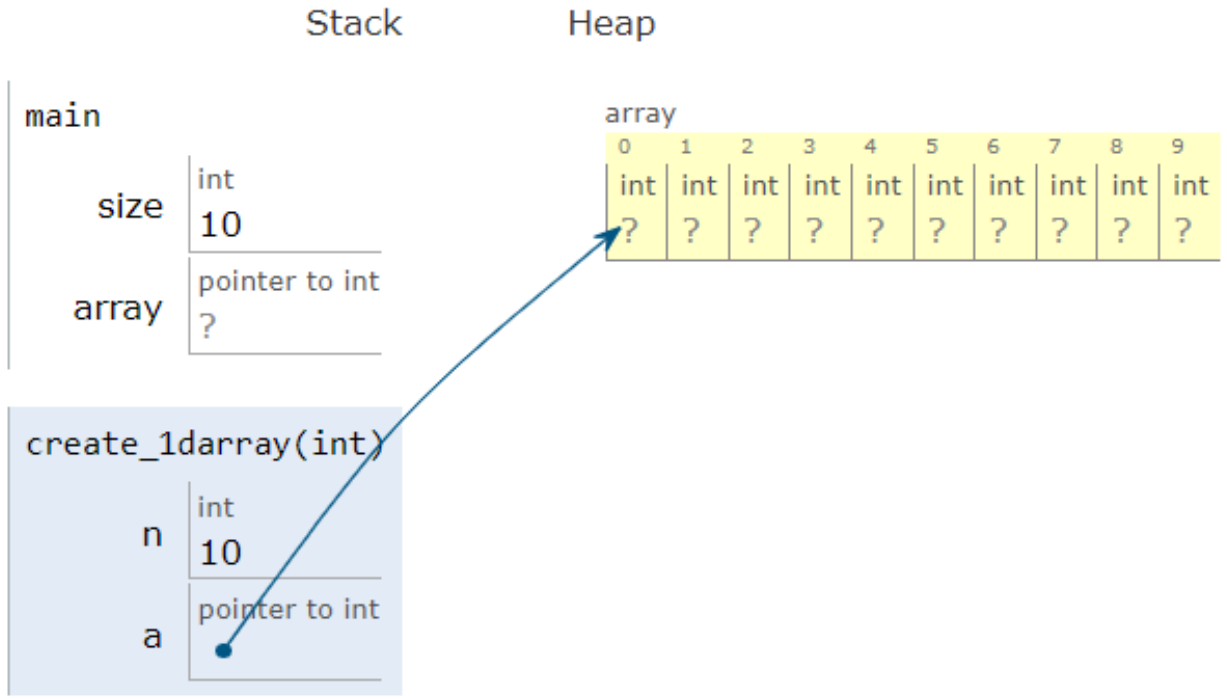


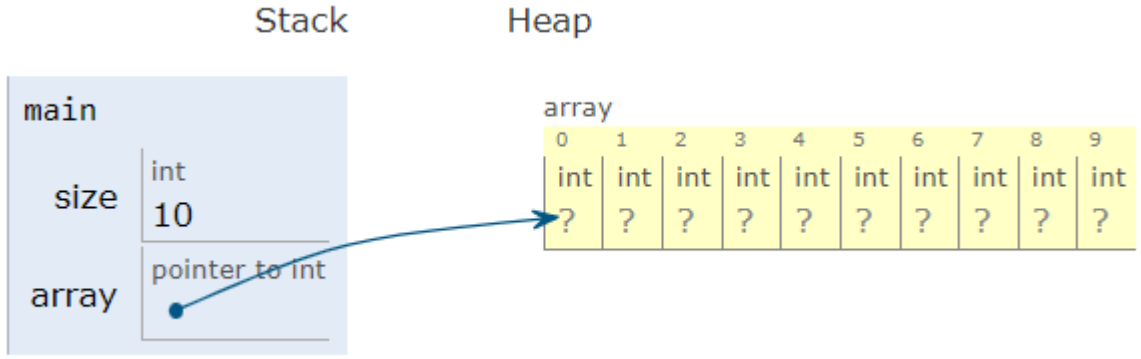
Exercise

- 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

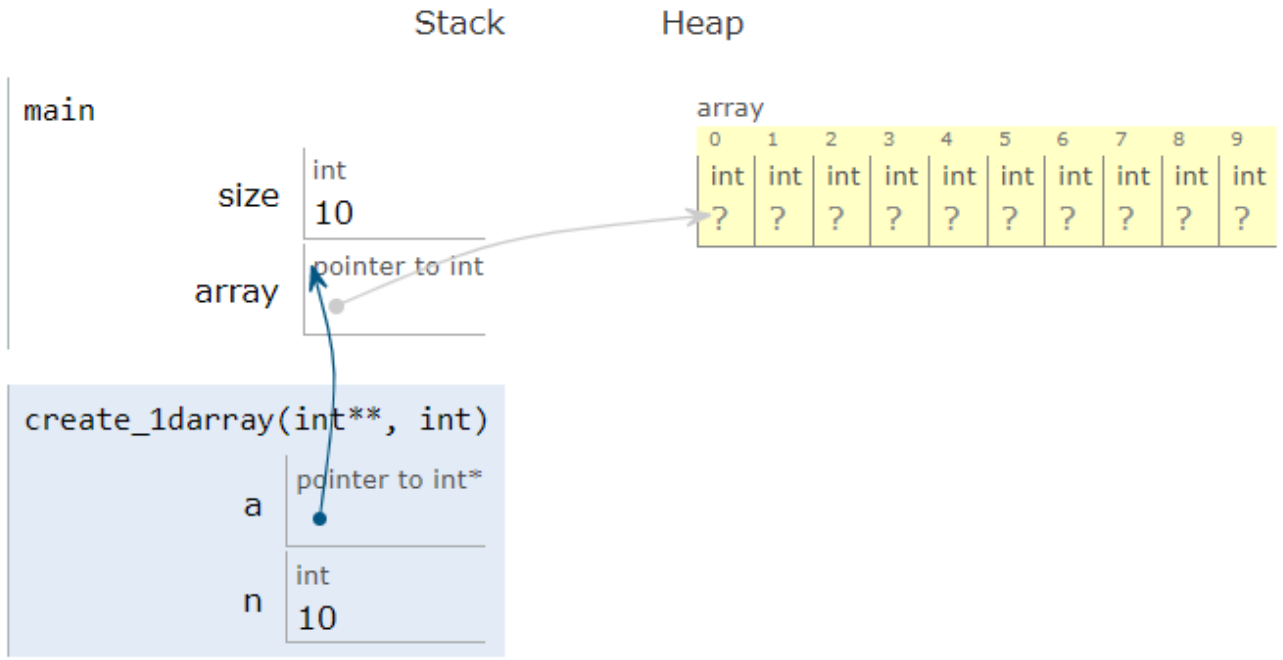
```
int main() {
    int *array;
    …
    array = create_1darray(size);
    …
}
int *create_1darray(int n) {
     int *a = new int [n];
     return a;
   }
```

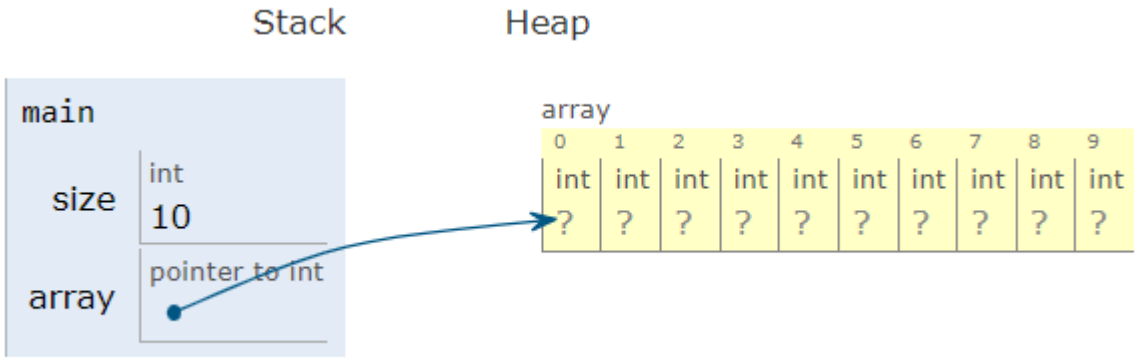




Create 1-D Array in Functions

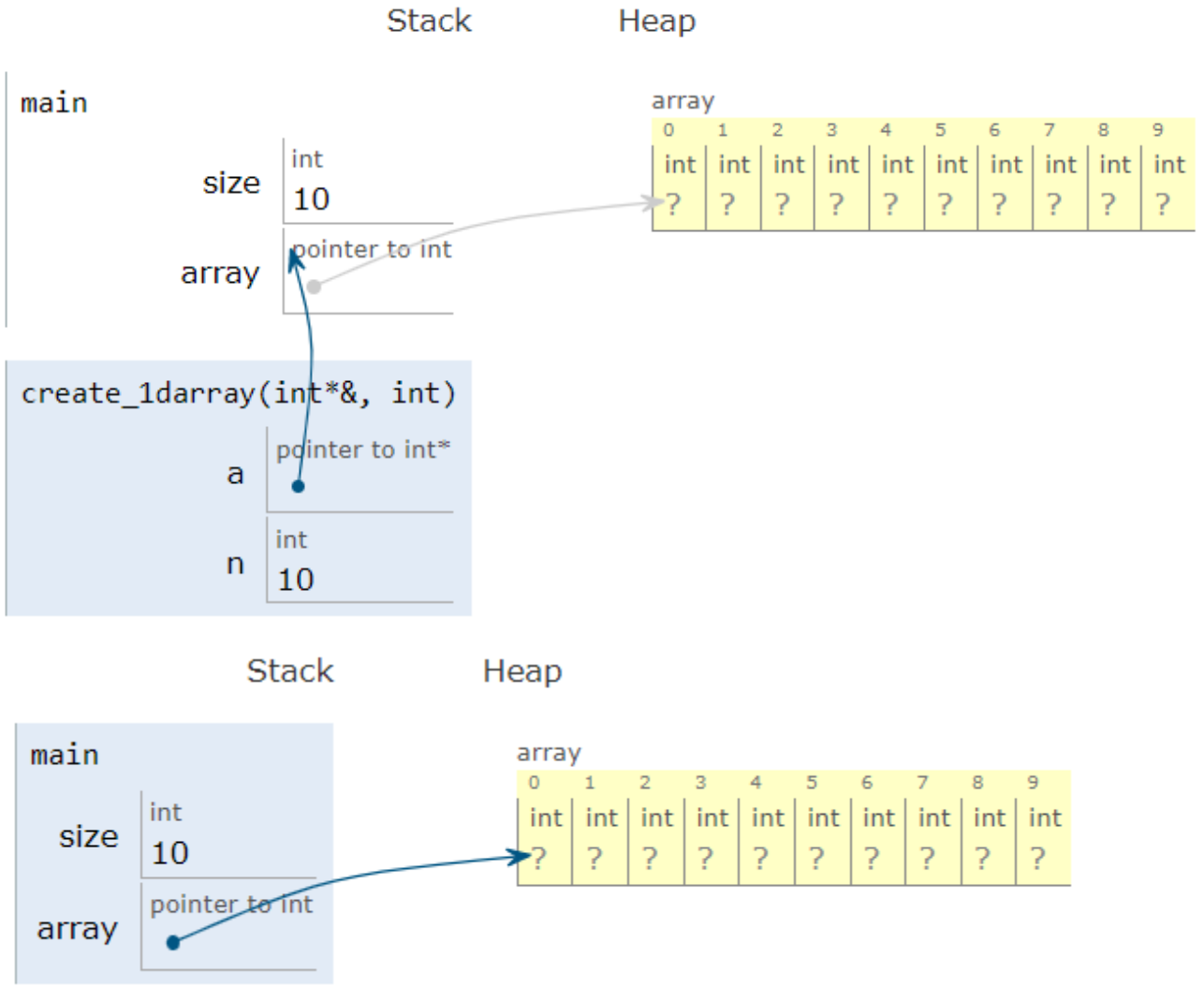
```
int main() {
   int *array;
   …
create_1darray(&array, size);
   …
}
void create_1darray(int **a, int n) {
    *a = new int [n];
}
```





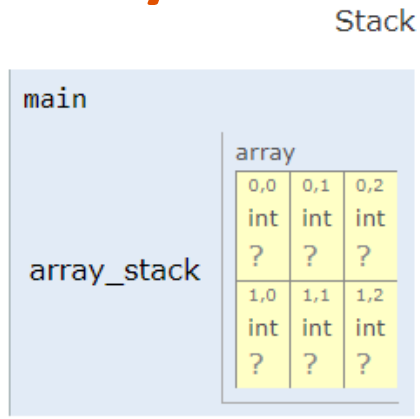
Create 1-D Array in Functions

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

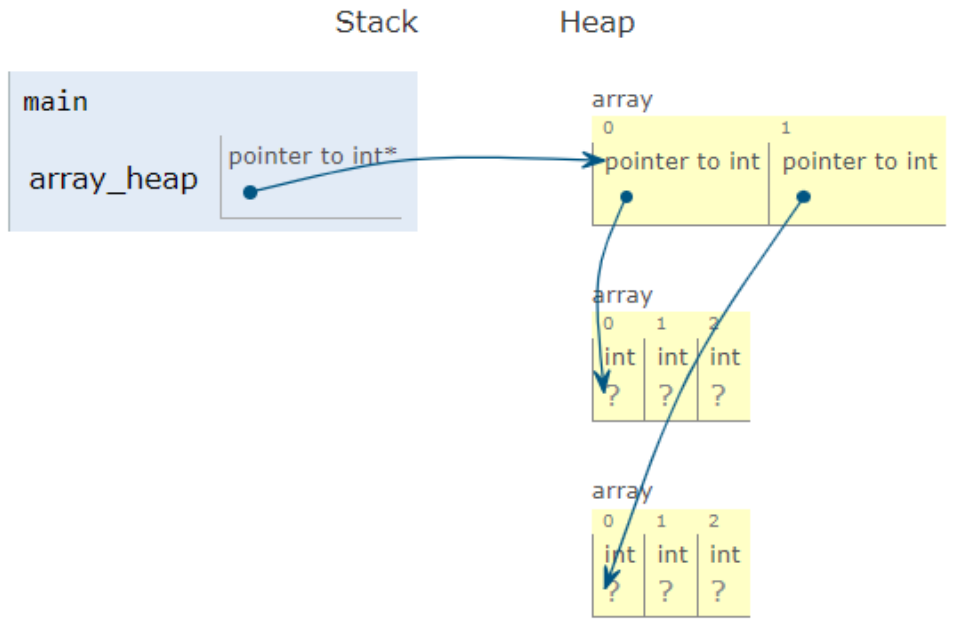


Static vs. Dynamic 2-D arrays…

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

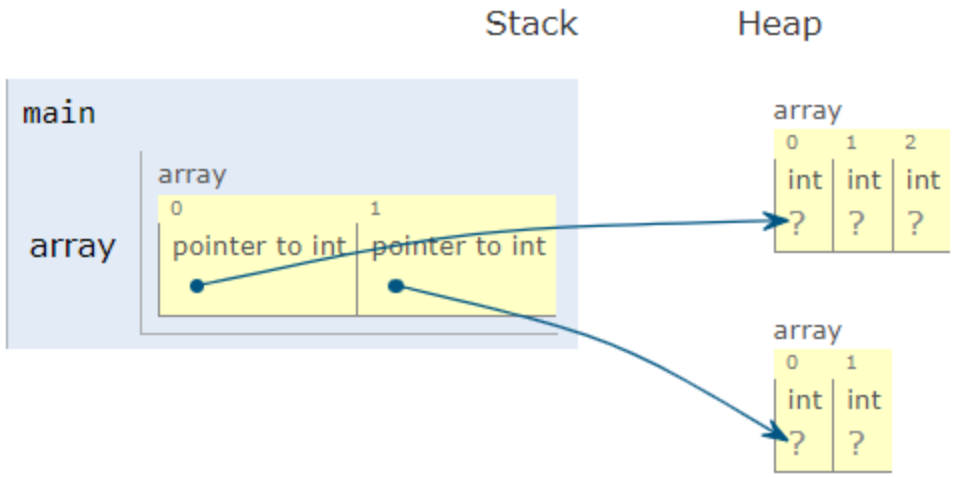


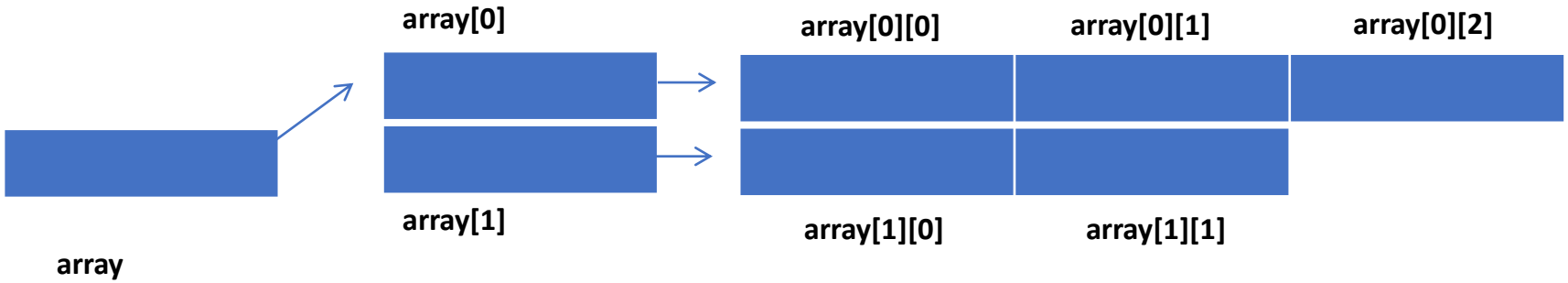

```
1  int main() {
2    int **array_heap = new int* [2];
3    for(int i = 0; i < 2; i++)
4      array_heap[i] = new int [3];
5
6    return 0;
7  }
```



Jagged Arrays

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





Passing a 2-D Array (Dynamic)

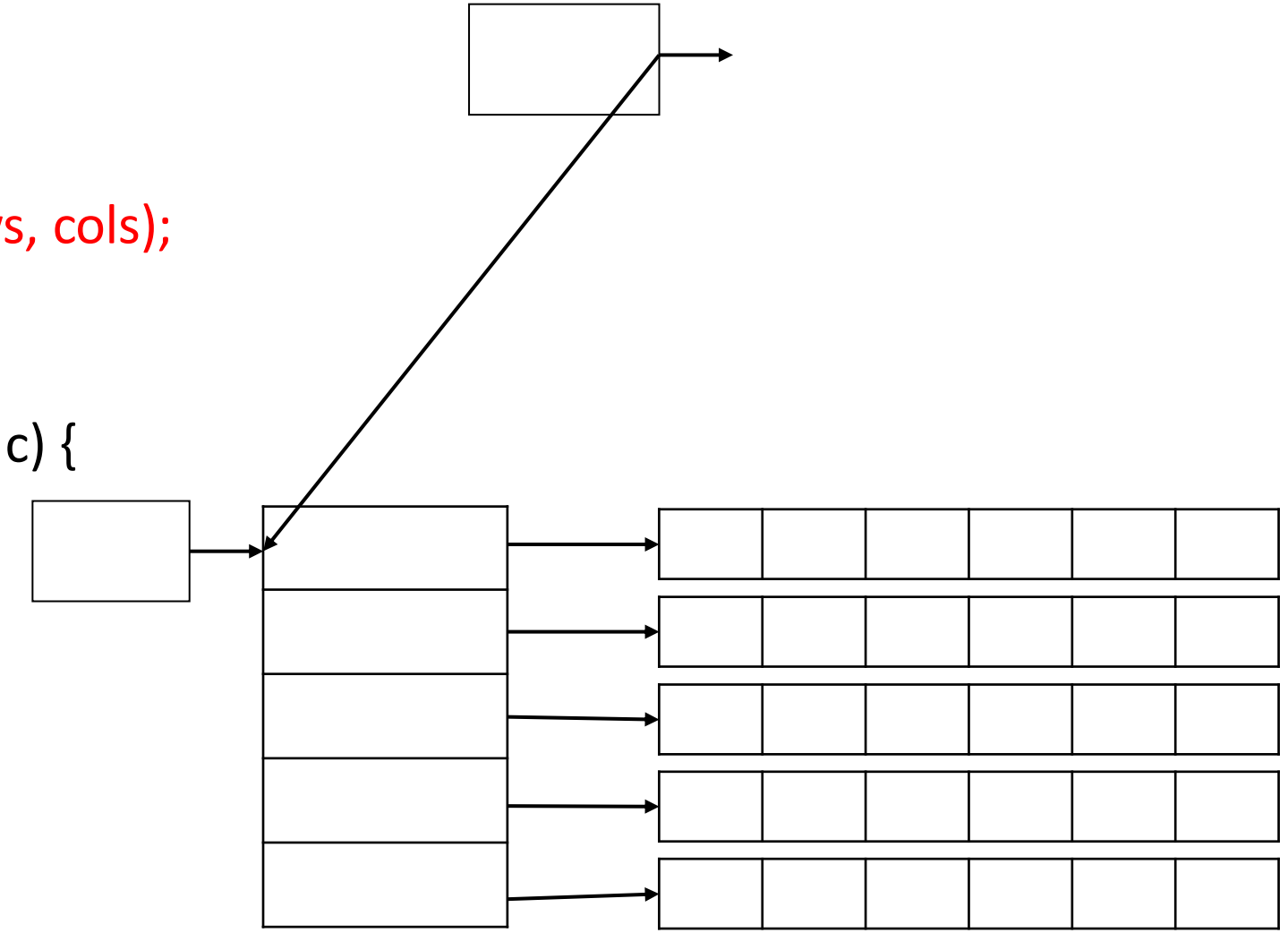
```
int main() {
    int **array;
    …
    pass_2darray(array, row, col);
    …
}
void pass_2darray(int *a[], int row, int col) {
     cout << “Array at zero: ” << a[0][0] << endl;
}
```

OR

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

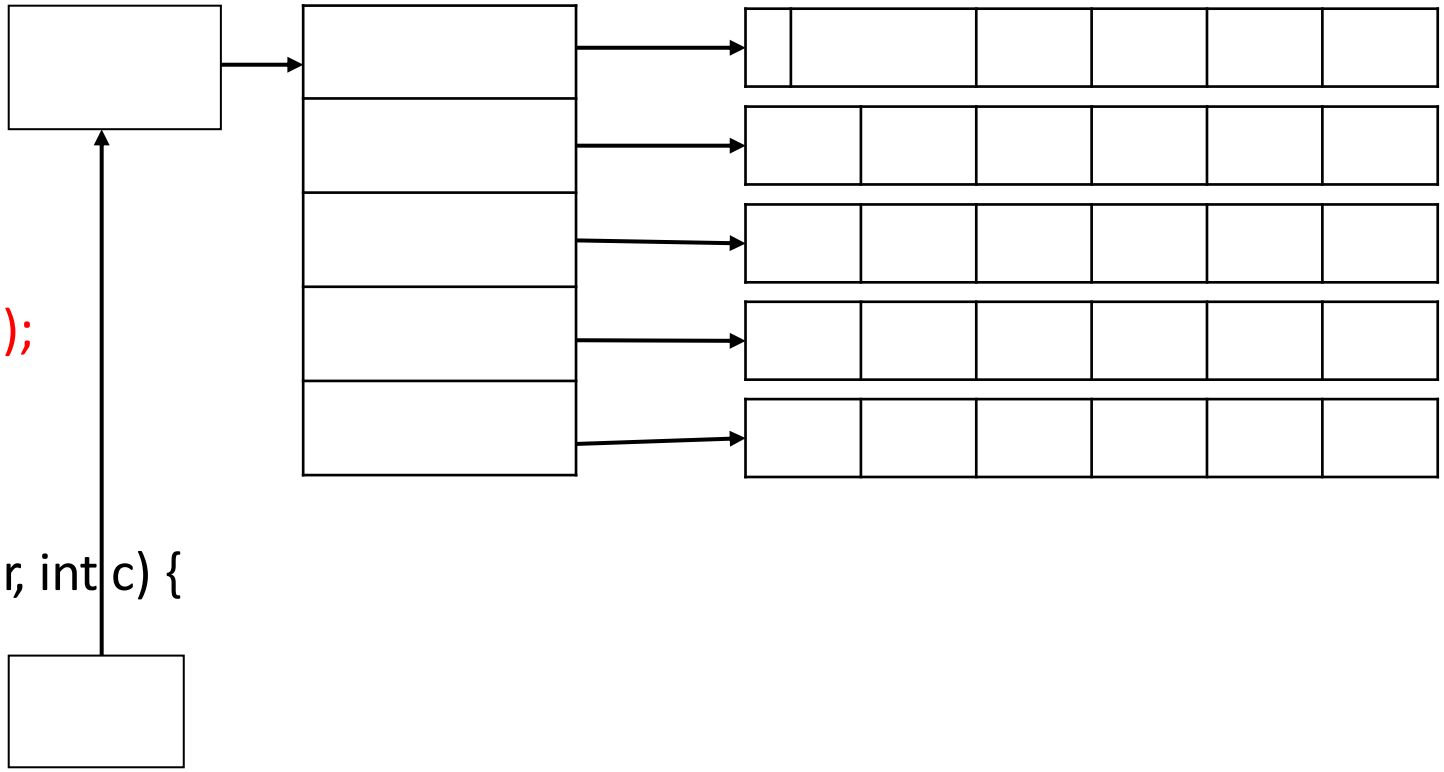
Create 2-D Array in Functions

```
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;
   }
```



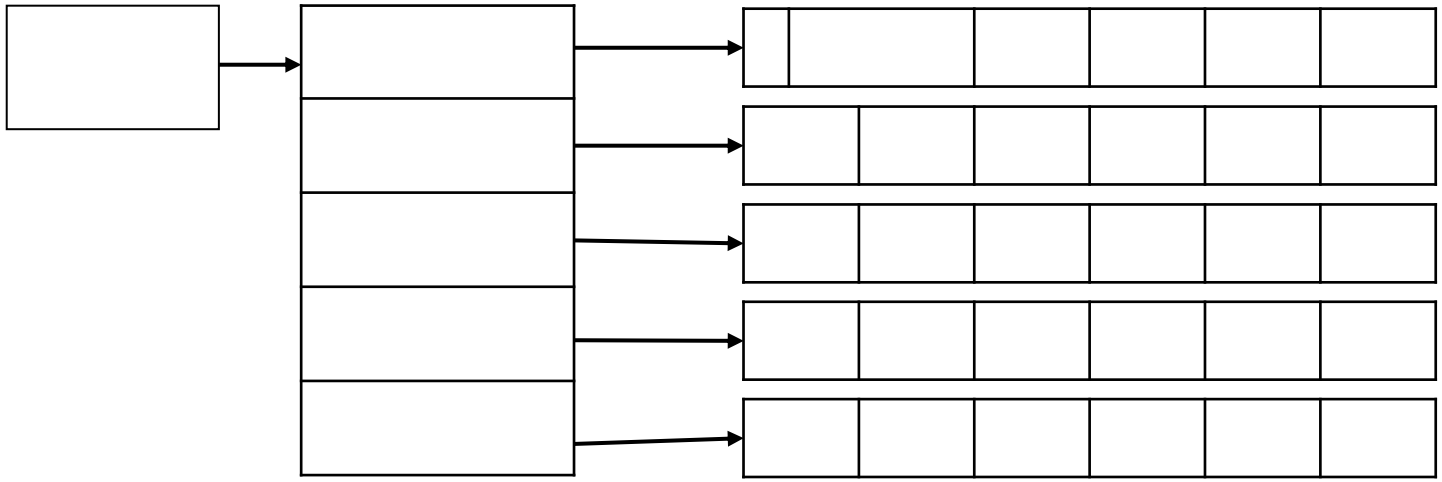
Create 2-D Array in Functions

```
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;
```