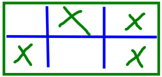
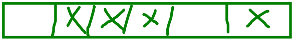
# quiz!

Consider the follow array declaration:



```
float grid[2][3];
```



Which of the following pairs of elements are adjacent in memory?

A. grid[0][2] and grid[1][2]

B. grid[0][2] and grid[1][0]

C. grid[0][1] and grid[1][0]

D. More than one pair of elements are adjacent

E. None of the pairs of elements is adjacent

# Returning an array from a function

returnArray.c:

```c
#include <stdio.h>
int * createArray(int size) {
    int a[size];  //declare an int array of specified size
    //...some initialization could happen here...
    return a;     //return the statically-allocated array
}
int main() {
    int *list = createArray(10);
    for (int i = 0; i < 10; i++) {
        printf("%d ", list[i]);
    }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic returnArray.c
returnArray.c: In function ‘createArray’:
returnArray.c:5:12: warning: function returns address of local variable
    5 |     return a;     //return the statically-allocated array
      |            ^
```

# Arrays and functions

- Arrays returned from functions are passed by address also; only a copy of the address is sent back to caller
    - But if the address is of an array that lives in the function's stack frame, the array won't survive after the function returns (the frame will be popped!)
- As a result, we can't expect to create an array that lives in a function's stack frame and then return it to the calling function
    - But we'll soon see a way to send back a new array from a function...

# Allocating a very large array

largeArray.c:

```c
#include <stdio.h>

int main() {
    int list[10000000];
    for (long i = 0; i < 10000000; i++) {
        printf("%d ", list[i]);
    }
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic largeArray.c
$ ./a.out
```

```
Segmentation fault (core dumped)
```

# Allocating a very large array

- Stack frames have a limited size
- On the last slide, we attempted to allocate an array within a function's stack frame, but the array was too large for the frame
    - A segmentation fault resulted

# Limitations of arrays allocated within a stack frame

- We've just seen that arrays allocated within a stack frame ("static allocation") have several limitations
    - Size of array is limited by size of stack frame
    - Arrays created within a called functions stack frame can't be accessed by calling function (since lifetime of array ends when called function returns)
    - Prior to C99, another limitation existed:
        - Needed to know size of array prior to run-time - couldn't ask for array of size `n` when `n` was a value input by user!
- To get around these limitations, we'll need *dynamic allocation*

# Dynamically-allocated memory

- Dynamically-allocated memory is located in a part of memory separate from the stack; it lives on "the heap"
- Dynamically-allocated memory lives as long as we like (until entire program ends)
  - We don't necessarily lose access to it when function call returns
    - This means we can return it to a calling function!
- Dynamically-allocated memory is not subject to size limitations based on stack frame size, since it's not part of the stack
- The size of a dynamically-allocated block of memory can be decided at run time

# Dynamically-allocated memory

- Dynamically-allocated memory solves lots of problems. . .
- But there is a catch: since it is not automatically reclaimed when function call ends, *we are responsible for telling system when we're through with this memory*
  - that is, we need to remember to *deallocate* it
  - allocated memory is not available to other programs/users until we deallocate it
  - failing to deallocate memory is the cause of "memory leaks"

# Dynamically-allocated memory

- To allocate memory, we can use a command named `malloc` (memory allocate) from `<stdlib.h>` (need to `#include`):

```
// allocate space for one int on heap
int *ip = malloc(sizeof(int));
// check if allocation succeeded
if (ip == NULL) { /*output error message*/ }
```

- After allocation with `malloc`, memory has not been initialized

```
// give dynamically-allocated int an initial value
*ip = 0;
```

# Dynamically-allocated memory

- When usage of dynamically-allocated int is complete, deallocate it using *free* command on address of the memory on the heap:

```
// notify system that we're through with heap int
free(ip);

// avoid accidental attempt to use this pointer
// to access the released space later
ip = NULL;
```

# Dynamically-allocated arrays

- To allocate an array with space for `n` items, express desired number of bytes via product of `n` and base type size:

```
int *a = malloc(sizeof(int) * n);
if (a == NULL) { /*output error message*/ }
```

- To access array items, use the usual square bracket notation:

```
a[0] = 0;
a[n-1] = 0;
```

- To deallocate the entire array of size `n`:

```
free(a); // no mention of array size needed here
a = NULL;
```

# Where should deallocation occur?

- Deallocation need not happen in same function where allocation occurred. . .
- . . . but *some* function needs to deallocate the block of memory!
  - Programmer's responsibility is to determine where deallocation will occur, and then ensure that it really does happen

# Now, a function that creates and returns an array!

```
returnDynAllocArray.c:
#include <stdio.h>
#include <stdlib.h>
int* createArray(int size) {
    int *a = malloc(sizeof(int) * size);  // declare array of size ints
    if (a == NULL) { return NULL; }       // exit if malloc failed
    //  ...array initialization really ought to happen here...
    return a;        // return the dynamically-allocated array
}
int main() {
    int *list = createArray(10);
    if (!list) { return -1; }    // abort program if function failed
    for (int i = 0; i < 10; i++) {
        printf("%d ", list[i]);
    }
    return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic returnDynAllocArray.c
$ ./a.out
0 0 0 0 0 0 0 0 0 0
```

# But. . .

- We forgot to free the dynamically-allocated memory!

# Better: remembering to release the memory

```c
returnAndFree.c:
#include <stdio.h>
#include <stdlib.h>
int* createArray(int size) {
    int *a = malloc(sizeof(int) * size);  // declare array of size ints
    if (a == NULL) { return NULL; }       // exit if malloc failed
    // ...array initialization really ought to happen here...
    return a;     // return the dynamically-allocated array
}
int main() {
    int *list = createArray(10);
    if (!list) { return -1; } // abort program if function failed
    for (int i = 0; i < 10; i++) {
        printf("%d ", list[i]);
    }
    free(list); //(why don't we use a here?)
    list = NULL; //not really needed; main is ending
    return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic returnAndFree.c
$ ./a.out
0 0 0 0 0 0 0 0 0 0
```