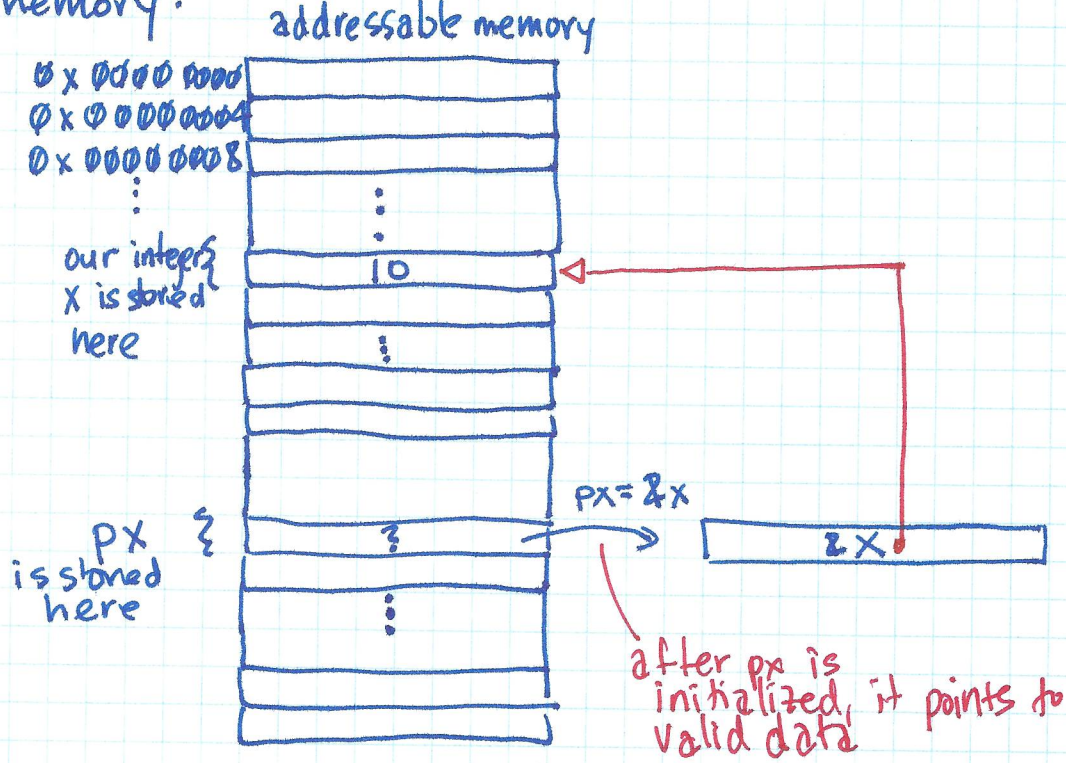
Stack/Queue Implementation Using Pointers

- recall that pointers are variables that contain addresses; that is, a pointer literally points to another variable:

ex.

```
int x = 10;  // C allocates memory (from
             // either stack or data
             // areas) to represent x,
             // initializing this memory
             // to the value of 10
```

in memory:



now consider:

```
int *px;  // C allocates memory to represent
          // px, but what px contains is
          // unknown — it is an uninitialized
          // pointer, and one must be careful
          // with it!
```

```
px = &x;  // px is ~~set~~ to point to x
```

Now, suppose we wish to change x to a value of −7. this can be done in two ways:

```
x = -7;  // straight forward!
```

or:

```
*px = -7; // here, * takes on another
          // meaning, that is, to
          // dereference px, accessing
          // whatever memory the pointer
          // points to
```

NOTE: in dereferencing a pointer, one might either read or write to the address the pointer contains.

Therefore, ALWAYS initialize your pointers so that they point to VALID MEMORY; if you attempt to write to an uninitialized pointer, your program will usually CRASH!!

Initializing Pointers with malloc()

- You can initialize your pointers by setting them to addresses of known variables (as above).

ex.

```
float y;

float *pfloat;
pfloat = &y;
```

ex.

```
char z;
char *p32;
p32 = &z;
```

but another way is to use malloc(·) which is the (heap) memory allocation function:

ex.

```
float *pf2 = (float *) malloc (sizeof (float));
char *p33 = (char *) malloc (sizeof (char));
```

malloc reserves an amount of memory (in bytes) given by the sizeof(·) argument, and returns a void pointer to this newly allocated memory

Q: • does everyone know how sizeof(·) works?

Passing by Pointer

• passing by pointer in function calls serves a couple of purposes:

① when we wish to modify the variable being passed in;

② when we wish to avoid copying significant amounts of memory (as can happen with pass-by-value (e.g., instead of passing in an entire array, simply pass in the pointer to the array)

e.g.,

```
int myfunc (int *px); // myfunc() prototype
:
int x = 3;
int y = myfunc (&x);
```

ex.

```
void sort_array (int *data, size_t N);
        :
int array_of_ints[] = { -3, 22, 6, 8, -7284, 83112, 29 };
size_t length = sizeof(array_of_ints) / sizeof(int);
sort_array (array_of_ints, length);
```

if we want avoid (or prevent) the alteration of a variable, we can use the const keyword in the function prototype — this gives us point ② without having to also have ①.

Stack w/ Pointers

we're using the same pseudo-code/algorithm, just better C code!

- our previous implementations of stacks & queues were based on global variables, which, while sometimes necessary, are not the most modular approach to writing software...
  - what if we want a self-contained stack or queue library?
  - what if we need multiple stacks & queues in the same application — having global variables makes for an ~~unweild~~ unwieldy situation!!

- we'll assume the stack will store integers, as before.
- let's create a stack variable type, as follows:

```
#include <stdlib.h>
#include <stdbool.h>
```
— allows us to use "bool", "true" and "false"

```
struct stack_struct
{
    int data[N];
    size_t top;
};
typedef struct stack_struct stack_t;
```

N should be defined using, for example, `#define N 1024`

- now, our algorithms can have the following prototypes:

```
bool stack_empty (stack_t *s);
void push (stack_t *s, int x);
int pop (stack *s)
```

NOTE THAT THESE VERSIONS MORE CLOSELY RESEMBLE OUR ORIGINAL PSEUDOCODE!!

- Q: can you describe the potential benefits of passing our stack by pointer?

— let's now look at the associated github project...