# CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

# Today's Goals

Subject Matter

- The Stack, review
- The Heap

Your Goals

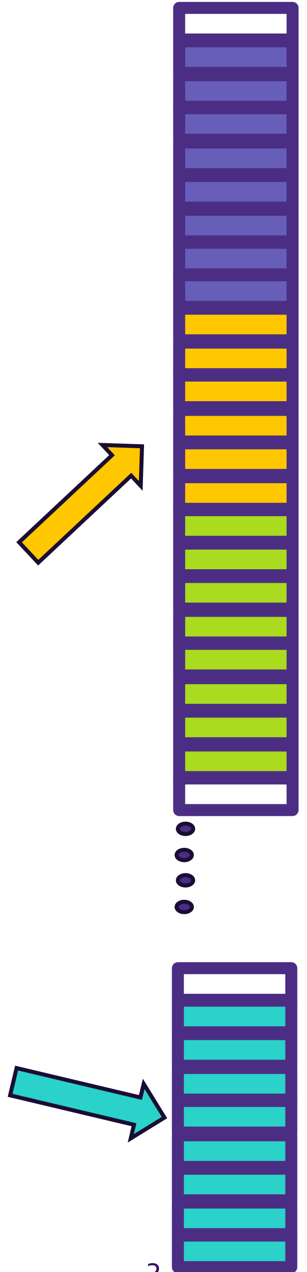
- Homework 3
- Pointer Puzzles

# Variable Scoping

- Variables (named things) accessible at certain times
- Scope and storage are related but not the same thing.

- Global variables
  - Scope: entire program
  - Violate encapsulation, but can be okay for truly global data (such as conversion factors)
- Static Global Variables
  - Scope: containing file/module
  - Can not be called from other files
- Static local variables
  - Scope: that function

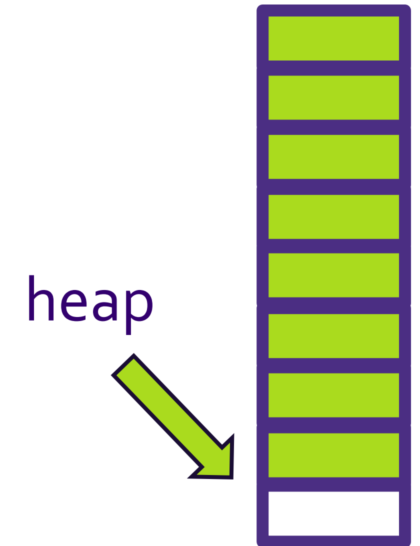
- Local variables
  - Scope: within the block



# Memory Management

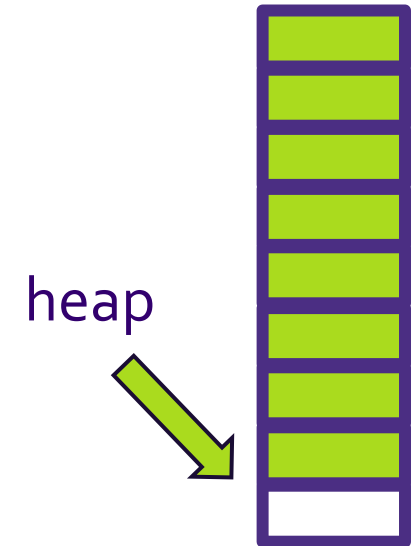
- All variables need a place to live in memory
  - Get '`allocated`' a physical space in memory (with an address)
  - `Sizeof` memory depends on what you are storing
  - Gets '`de-allocated`' when you no longer need the memory
- 'Stack' holds current instructions, each function in a frame
  - This is where active function variables are stored – arguments and local variables
- **'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)**
- The heap and stack grow dynamically. Meet in the middle ?= 'out of memory' error

# The Heap



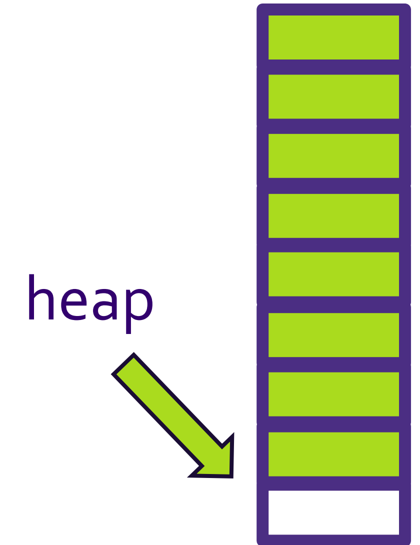

- Provides user (programmer) flexible space
- Allocated at run-time with current space requirements
  - Could be calculated during the run
- Persists until specifically de-allocated (`free` –ed)
  - Exists outside of stack management, so doesn't disappear with current function
- `void* malloc (size_t size);`
  - `size_t` is an unsigned integral, `size` is a number of bytes
  - Returns `void*` : an untyped pointer
  - If it fails, returns `NULL`
- **Always check for NULL** before you use the pointer: is a reason to exit the program with a failure code.

# Using malloc




- Provides user (programmer) flexible space
- Allocated at run-time with current space requirements
  - Could be calculated during the run
- Persists until specifically de-allocated (`free` –ed)
  - Exists outside of stack management, so doesn't disappear with current function
- **`void* malloc (size_t size);`**
  - `size_t` is an unsigned integral, `size` is a number of bytes
  - Returns `void*` : an untyped pointer
  - If it fails, returns `NULL`
- **Always check for NULL** before you use the pointer: is a reason to exit the program with a failure code.

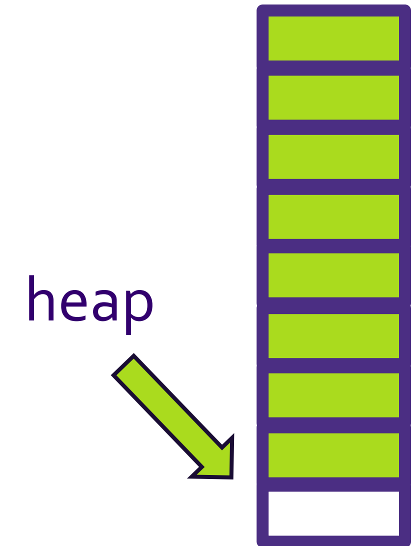
# Pattern of malloc calls




- `void* malloc (size_t size);`
- `T* newmem = (T*) malloc (e* sizeof(T));`
  - `T` : a data type
  - `e` : number of elements (often 1!)
  - `sizeof` : calculates size of data type, since the user may not know it
    - More portable than assuming
  - `(T*)` : casting the untyped pointer to a pointer to a T
- **Always check for NULL** before you use the pointer: is a reason to exit the program with a failure code.

```
int *var = (int*)malloc(1*sizeof(int));
*var = 255;
```

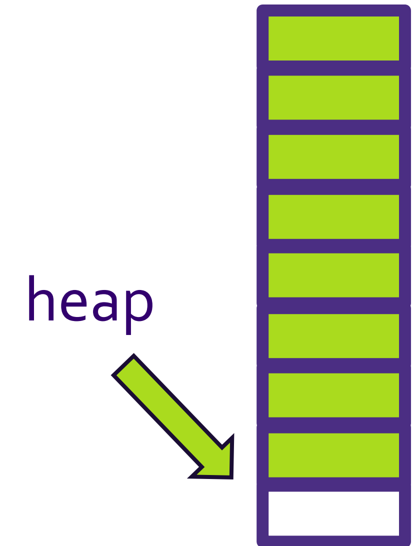
# Initializing memory




- malloc allocates space, but doesn't set value of bits stored there
- **You must initialize the values manually**
- **Or use calloc, but its slower**
- `void* calloc (size_t num, size_t size);`
  - `size_t` is an unsigned integral,
    - `num` is how many elements in an array
    - `size` is a number of bytes in one element of the array
  - Returns `void*` : an untyped pointer
    - **Initializes all values in the memory to zero**
  - If it fails, returns `NULL`

```
int *arr = (int*)calloc(25*sizeof(int));
arr[20] == 0;
```

# Re-allocating memory



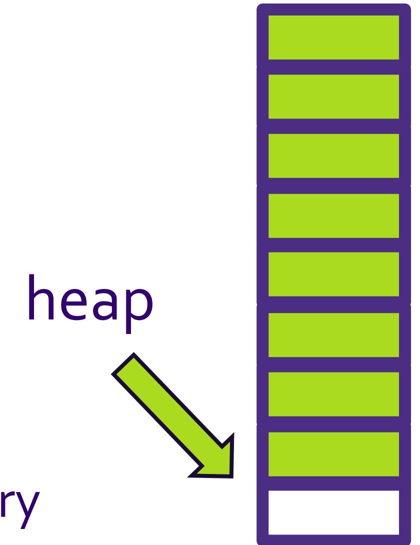

- Must know how big an array is before you allocate it
- But – you can change the size with `realloc` and `ptr`.
- **`void* realloc (void* ptr, size_t size);`**
  - `size_t` is an unsigned integral, `size` is a number of bytes
  - `ptr` is an existing pointer.
  - Returns `void*` : an untyped pointer
    - **Initializes min(old_size, new_size) with values from the ptr memory**
    - **Reuses memory, or moves it: de-allocates old memory as appropriate**
  - If it fails, returns `NULL`

```
int *arrbig = (int*)realloc(arr, 30*sizeof(int));
arrbig[20] == 0;
Arrbig[29] == ??;
```

# Halfway through with memory management

- Can allocate memory of any size, and have it persist forever
  - Could allocate an array and keep using it, independent of stack
- However, with finite memory, we don’t want to keep allocating more memory without freeing some up for re-use
- Jave solution: no explicit free, but garbage collector finds un-reachable objects and reclaims their space.
- C solution: user (programmer) explicitly free’s an objects space
  - This is hard!

# Freeing memory




- Releases previously allocated memory for re-use
- `void free (void* ptr);`
  - `void*` is an untyped pointer, `ptr` is the address of previously allocated memory
  - Returns `void` : nothing
  - If it fails, returns `NULL`
- If `ptr` does not point to memory previously allocated by malloc, calloc, or realloc: behavior is undefined.
- If `ptr` is `NULL` does nothing
- **NOTE:** The value stored at `ptr` doesn't change – it just now is a dangling pointer.
  - Set `ptr` to `NUL`, or use `free` right before `ptr` goes out of scope.

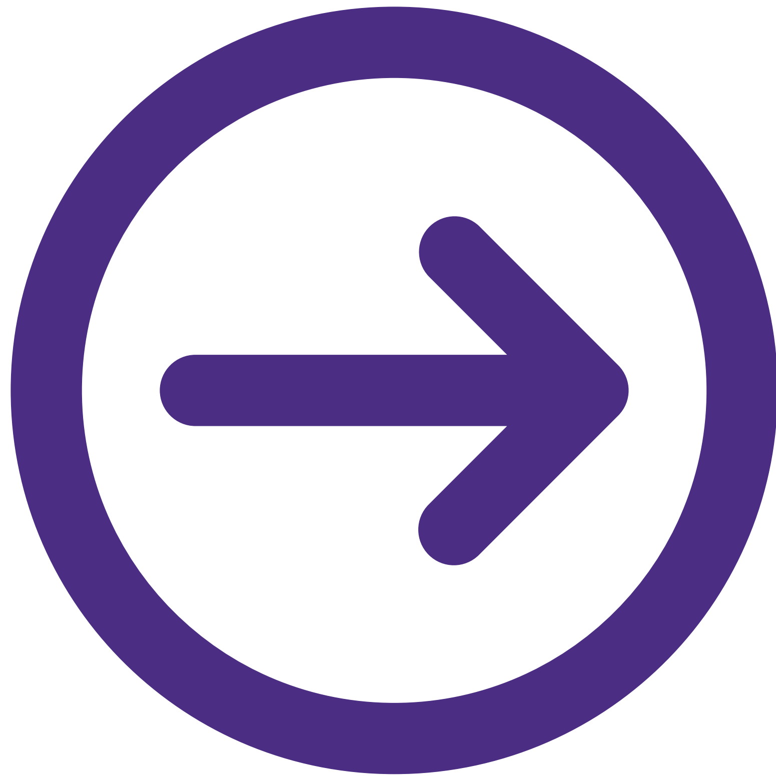
# Examples

```
int *p = (int*)malloc(sizeof(int));
p = NULL; /* LEAK! - lost address */
int *q = (int*)malloc(sizeof(int));
free(q);
free(q); /* Best case: crash */
int *r = (int*)malloc(sizeof(int));
free(r);
int *s = (int*)malloc(sizeof(int));
*s = 19;
*r = 17; /* Best case: crash */
```

Challenge with multiple functions:

- If you allocate in a function, and then exit, who frees?
  - better send the address back to another function
- If you get a pointer as an argument, can you free it?

You must pay attention to where you allocate and de-allocate.

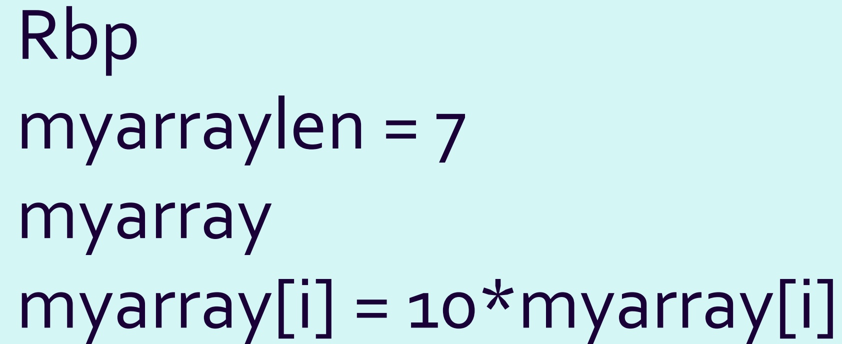


# Rules

- For every run-time call to `malloc` there should be one runtime call to `free`
- If you "lose all pointers" to an object, you can't ever call free (a leak)!
  - Note: It's possible but rare to use up too much memory without creating "leaks via no more pointers to an object"
- Think hard before re-assigning a pointer; where is it pointing?
- If you "use an object after it's freed" (or free it twice), you used a dangling pointer!

# Array behaviors

myarray connects back to the mainarray allocated in main. Will change memory in that location.




len = 7

mainarray = ~~??~~ HEAP

mainarray[i]=i

```c
void arrfunc (int *myarray, int myarraylen) {
  for(int i = 0; i < myarraylen; i++) {
    printf("Value of myarray[%d] is: %d \n", i, myarray[i]);
  }
  for(int i = 0; i < myarraylen; i++) {
    myarray[i] = 10*myarray[i];
  }}

int main() {
  int len = 7;
  int *mainarray;
  mainarray = (int*) malloc (len * sizeof(int));

  for(int i = 0; i < len; i++) {
    mainarray[i] = i; }
  arrfunc (mainarray, len);
  int *ind = mainarray;
  for(int i = 0; i < len; i++) {
    printf("Value of myarray[%d] is: %d \n", i, *ind);
    ind++;
  }
  //free (mainarray);
  return 0;  // return all successful
}
```

# Valgrind

**$valgrind *program arguments***

**$valgrind --leak-check=full ./dangling**

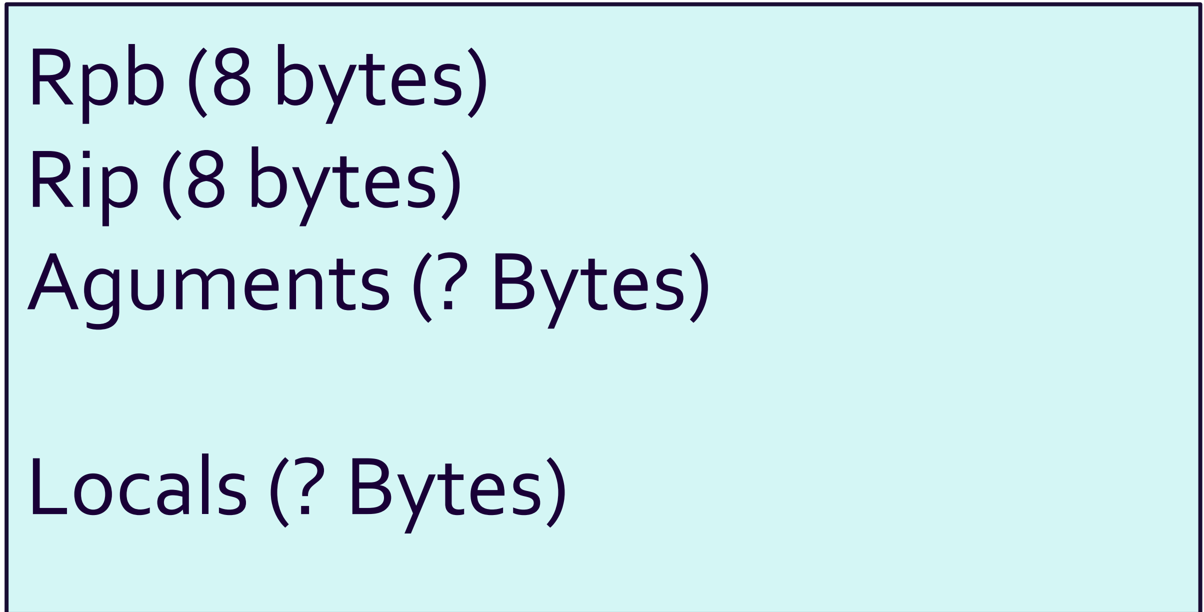
- Analyzes memory usage.
  - Catches pointer errors during execution
  - Prints summary of heap usage
  - including details of memory leaks

**$valgrind --leak-check=full ./arraydynamic**

- Good idea to always check a program before you call yourself done

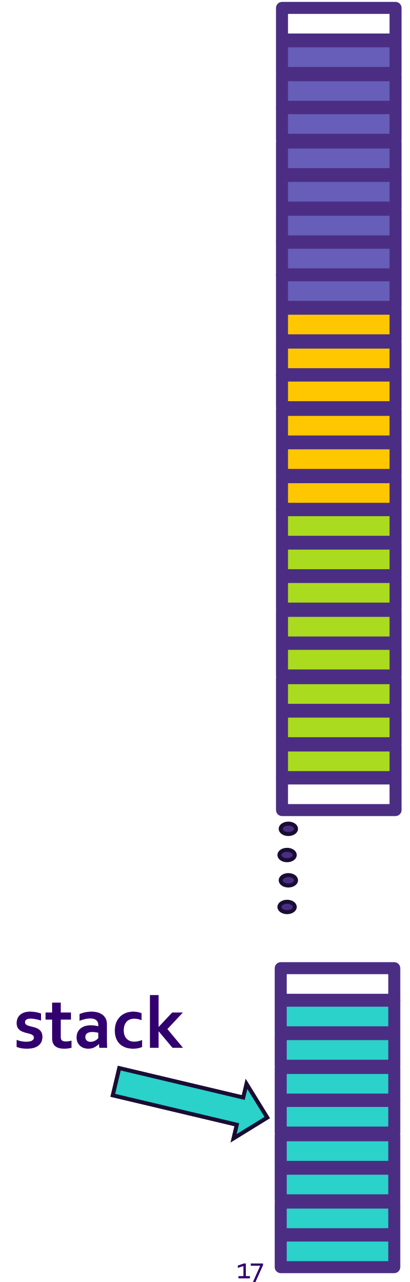
# BACKUPS

# The Stack (memory allocation)



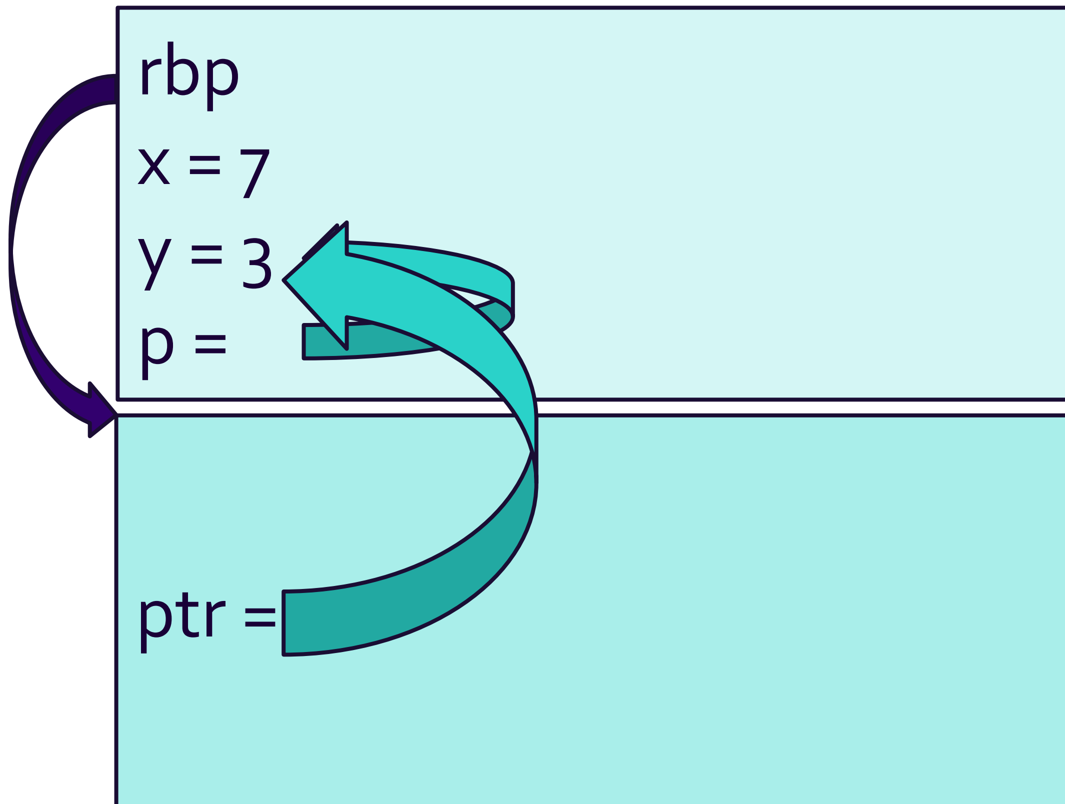

Callee frame

Caller frame



# Dangling Pointers

Pointers hold address, don't know if memory is allocated

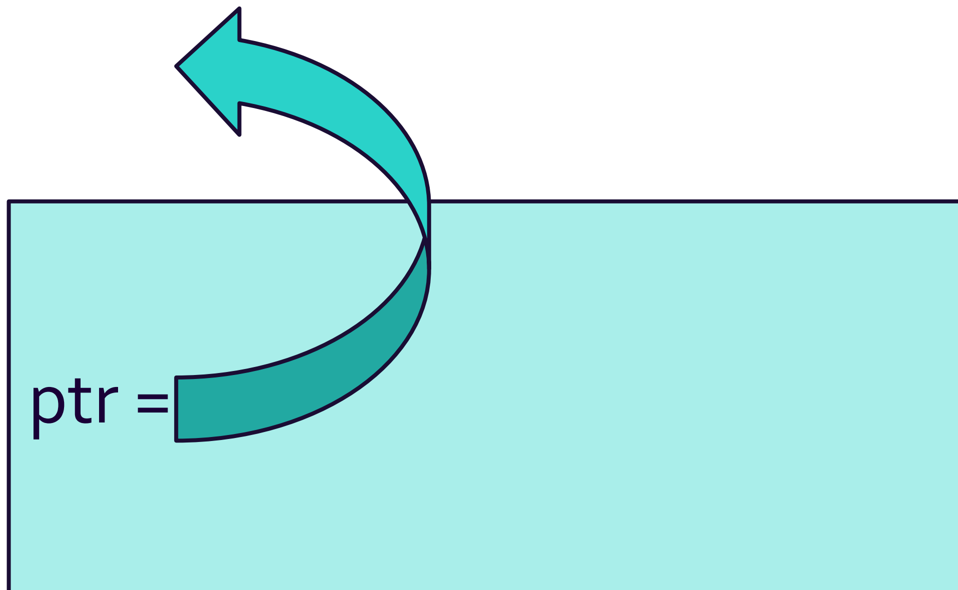



```c
int* f(int x) {
  int *p;
  int y = 3;
  p = &y; /* ok */
  return p;
}

int main(int argc, char **argv) {
  int* ptr;
  ptr = f(7);
  printf ("in main: *p=%d\n", *ptr);
}
```

# Dangling Pointers 2

Pointers hold address, don't know if memory is allocated




```c
int* f(int x) {
  int *p;
  int y = 3;
  p = &y; /* ok */
  return p;
}

int main(int argc, char **argv) {
  int* ptr;
  ptr = f(7);
  printf ("in main: *p=%d\n", *ptr);
}
```

# Puzzle: What prints?

```
int main(int argc, char **argv) {
  char ant[4] = "bed";
  int x[2];
  *x = 6;
  x[1] = 7;
  int y = 4;
  int *z = &y;
  *z = *x;
  printf("%d %d %d %s\n", *x, x[1], y, ant);
  mystery(ant, x + 1, y);
  printf("%d %d %d %s\n", *x, x[1], y, ant);
}
```

```
#include <stdio.h>

void mystery(char *a,int
*b,
  int c) {
  int *d = b - 1;
  c = *b + c;
  *b = c - *d;
  *d = *b - *d;
  a[2] = a[b - d];
}
```