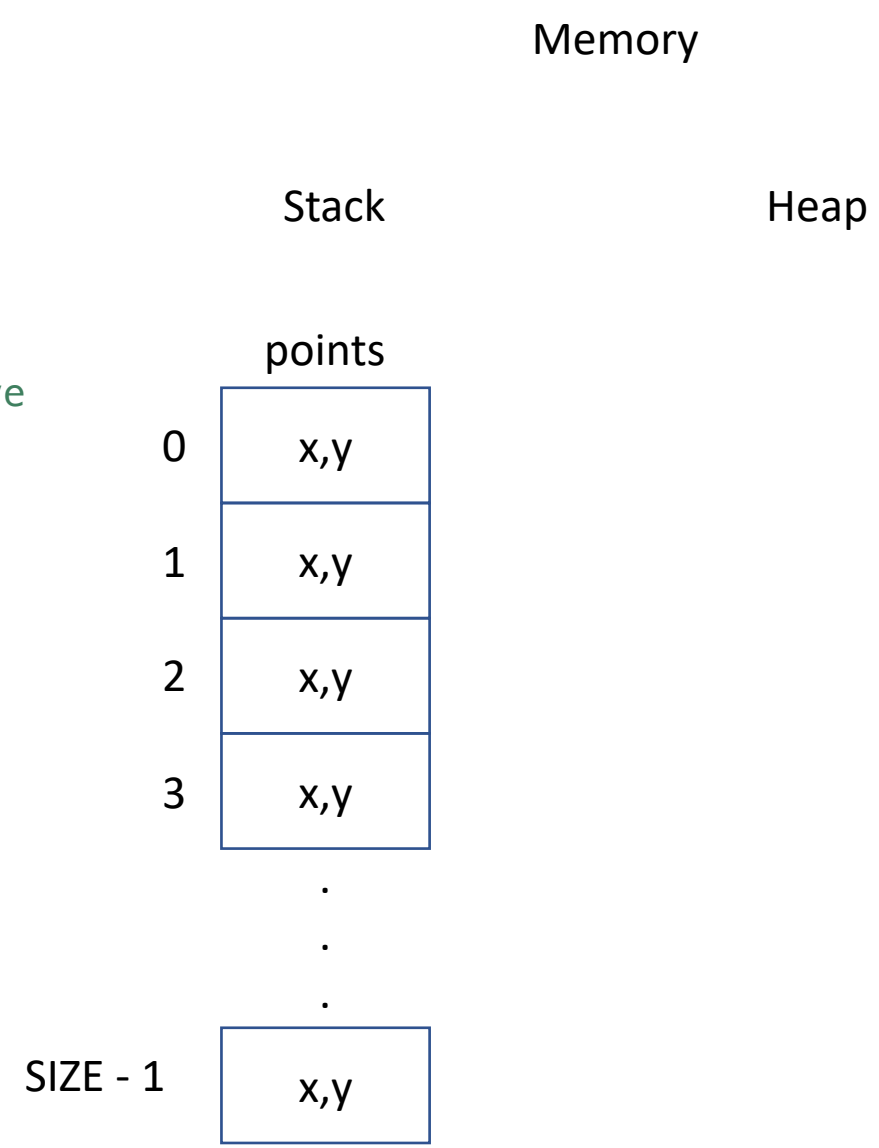
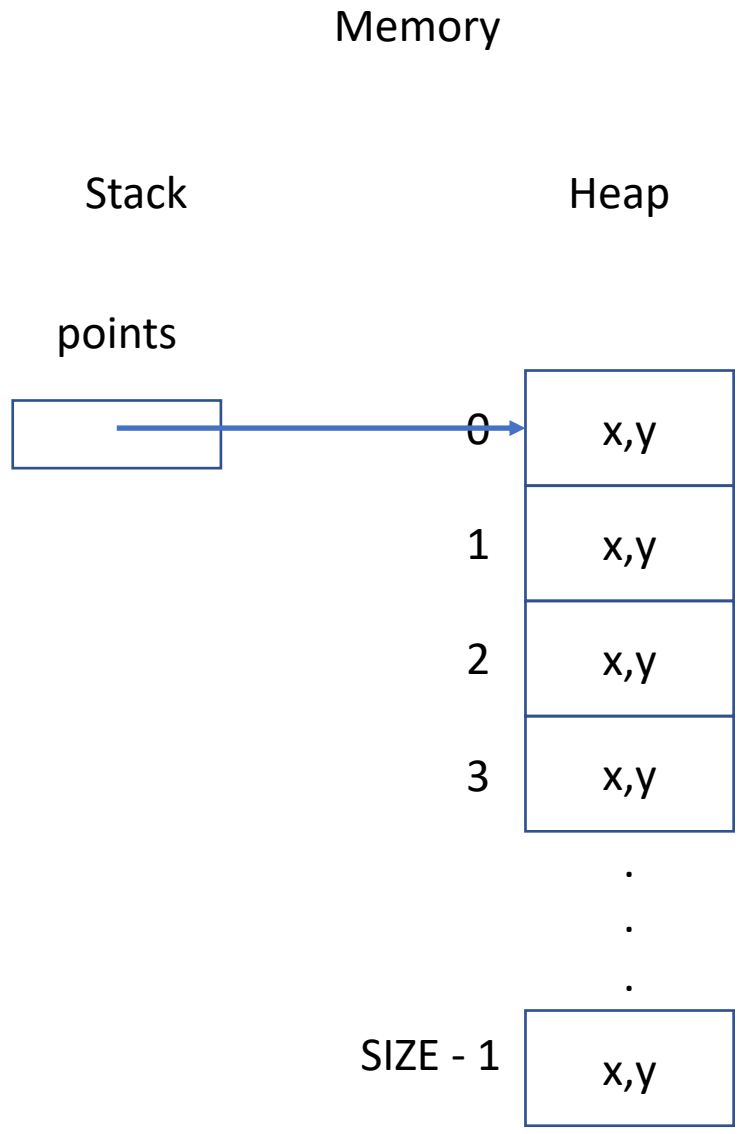
```c
#include <stdio.h>
#include <stdlib.h>


//two ways to define constants
#define MAX 100
const int SIZE = 10;



// A struct with two integer values.
struct point {
  int x;
  int y;
};
```

```c
// pt is passed by value (copied) as a parameter,
// so changes made here are not reflected in the original
void addOneByValue(struct point pt){
  pt.x++;
  pt.y++;
}


// pt is a pointer, so the struct is passed by reference.
// Changes made here are changes in the original.
// To access items in a struct pointed to by pt, use -> notation
void addOneByReference(struct point *pt){
  pt->x++;
  pt->y++;
}


// the struct is passed by reference, but no changes are made
void printPoint(char *label, struct point *pt){
  printf("%s (%d, %d)\n", label, pt->x, pt->y);
}
```

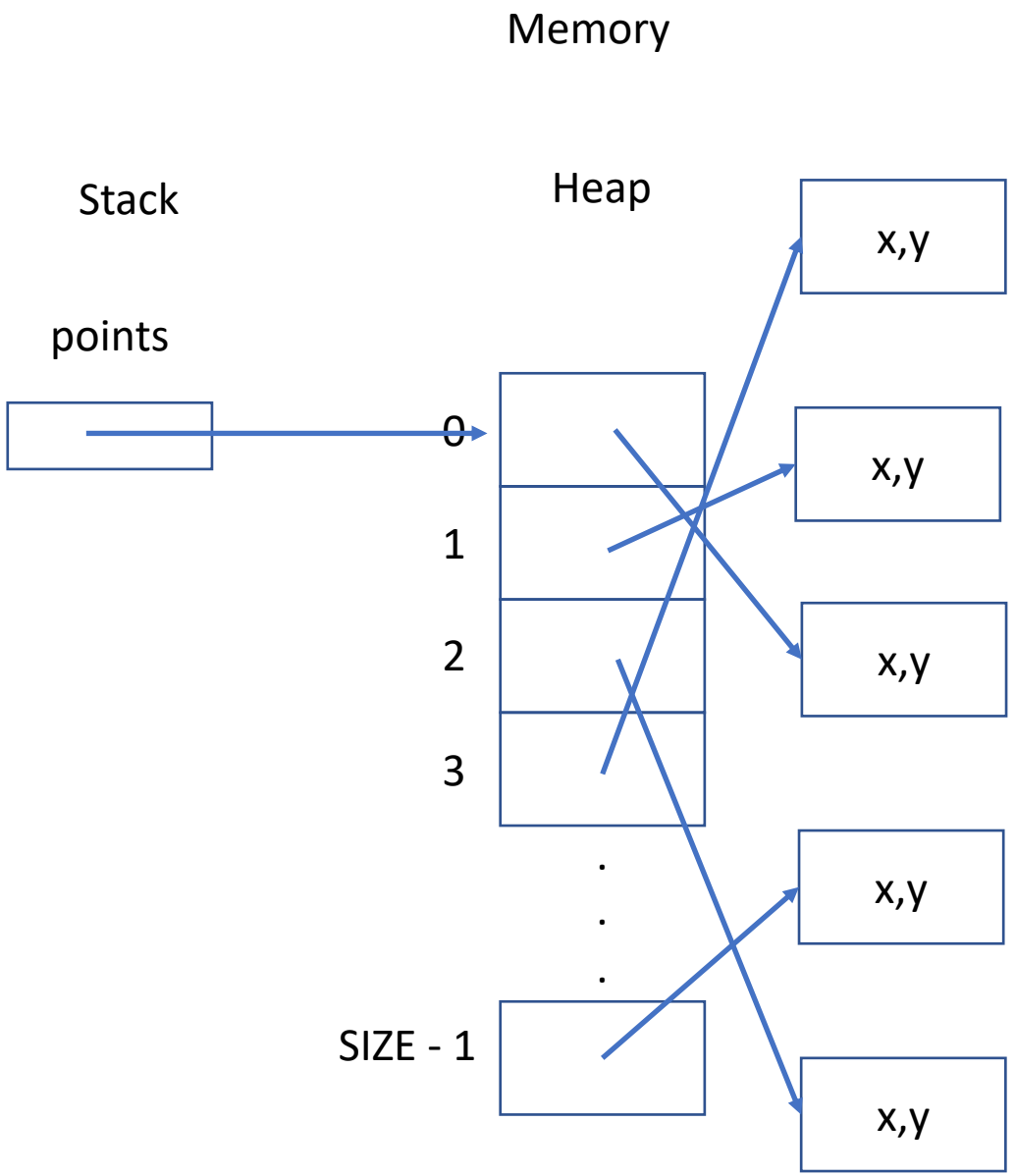
```c
//Create and print an array of points:
// This array is static, it is ALL on the stack
// It will no longer be useful after the method returns.
void staticArrayTest(){
  printf("staticArrayTest()\n");
  struct point points[SIZE];


  for(int i = 0; i < SIZE; i++){
    //rand() produces a number between 0 and RAND_MAX inclusive
    points[i].x = rand() % MAX;
    points[i].y = rand() % MAX;
  }


  for(int i = 0; i < SIZE; i++){
    //make enough room to hold the label
    char label[MAX];
    sprintf(label, "points[%3d]:", i);
    //pass the address of the struct to printPoint
    printPoint(label, &points[i]);
  }

  printf("\n");
}
```



```c
// Create and print an array of points:
// In this case the array is on the heap and is enough room for
// the structs to be lined up contiguously in memory.
// Each struct exists and has values.
// The entire array must be released when the program is done with it.
void dynamicArrayTest(){
  printf("dynamicArrayTest()\n");
  // Note the type: a pointer to a struct.
  // This is a memory address where the array begins.
  struct point *points;
  points = (struct point *)malloc(SIZE * sizeof(struct point));


  for(int i = 0; i < SIZE; i++){
    points[i].x = rand() % MAX;
    points[i].y = rand() % MAX;
  }


  for(int i = 0; i < SIZE; i++){
    char label[MAX];
    sprintf(label, "points[%3d]:", i);
    printPoint(label, &points[i]);
  }

  //Free the memory used by the array
  // never use this again afterwards
  free(points);

  printf("\n");
}
```

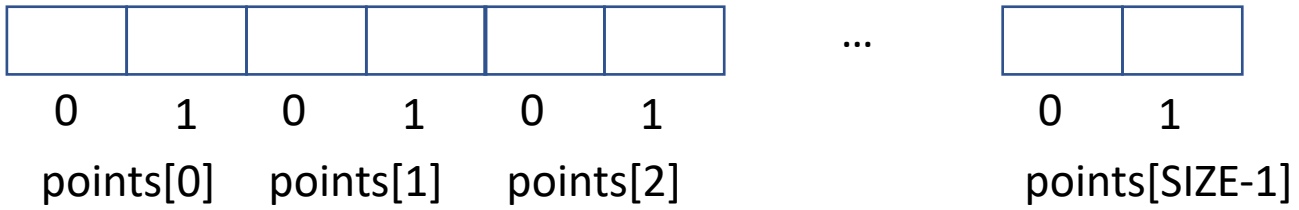


```c
// Create and print an array of points:
// This is an array of pointers to structs.
// Each struct is allocated separately, so must be freed separately.
// This is most closely models Java
void dynamicArrayOfPointerTest(){
  printf("dynamicArrayOfPointerTest()\n");
  // Note the type: a pointer to pointers of structs.
  struct point **points;

  // Create the array of pointers
  points = (struct point **)malloc(SIZE * sizeof(struct point *));

  for(int i = 0; i < SIZE; i++){
    //create a struct
    points[i] = (struct point *)malloc(sizeof(struct point));
    points[i]->x = rand() % MAX;
    points[i]->y = rand() % MAX;
  }

  for(int i = 0; i < SIZE; i++){
    char label[MAX];
    sprintf(label, "points[%3d]:", i);
    printPoint(label, points[i]);
  }

  for(int i = 0; i < SIZE; i++){
    // Once we are done with it, free it up.
    free(points[i]);
  }

  //Free the entire array
  free(points);

  printf("\n");
}
```



```c
void array2DTest(){
  printf("array2DTest()\n");
  // All of these values are contiguous in memory
  // A 2D array is a 1D array with syntax shortcuts
  // Completely unlike Java
  int points[SIZE][2];


  for(int i = 0; i < SIZE; i++){
    points[i][0] =  rand() % MAX;
    points[i][1] =  rand() % MAX;
  }


  for(int i = 0; i < SIZE; i++){
    printf("(%d, %d)\n", points[i][0], points[i][1]);
  }
  printf("\n");
}
```

Stack



```c
//This is equivalent to array2DTest
void array1DTest(){
  printf("array1DTest()\n");
  int points[SIZE*2];


  for(int i = 0; i < SIZE*2; i++){
    points[i] =  rand() % MAX;
  }

  for(int i = 0; i < SIZE; i++){
     printf("(%d, %d)\n", points[2*i], points[2*i+1]);
  }
  printf("\n");
}
```

Stack