The Heap and Structs
CSE 333 Summer 2020

Instructor:  Travis McGaha

Teaching Assistants:
Jeter Arellano  Ramya Challa  Kyrie Dowling
Ian Hsiao  Allen Jung  Sylvia Wang
About how long did Exercise 2 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn’t submit / I prefer not to say
Administrivia

- ex0 grades released, ex3 released today
  - Regrade requests: open 24 hr after, close 72 hr after release

- We recommend doing the extra exercises
  - Also, can Google for “C pointer exercises”
  - You MUST master pointers quickly, or you’ll have trouble with the rest of the course (including hw1)

- hw0 due tonight before 11:59 pm (and 0 seconds)
  - Git: add/commit/push, then tag with hw0-final, then push tag
    - Then clone your repo somewhere totally different and do git checkout hw0-final and verify that all is well
Yet More Administrivia (sorry)

- Exercise grading – Gradescope abuse
  - Grading score is an overall evaluation: 3/2/1/0
  - Then additional ±0 rubric items as needed
    - These are a quick way of communicating “why” – reasons for deductions or comments about your solution
    - Allows us to be more consistent in feedback
    - The -0 “score” is just because that’s how we have to use Gradescope to handle feedback notes – it does not contribute to “the points”
hw1 due Thursday, 7/09 11:59 pm

- You **may not** modify interfaces (.h files)
- But **do** read the interfaces while you’re implementing them(!)
- **New this quarter**: short answer questions in README.md
- Suggestions:
  - Make sure you understand the diagrams in the specification and draw box and arrow diagrams!
  - If you are stuck, take a break. When you come back, **scrutinize** your code.
  - Have more fun, less anxiety: pace yourself and make steady progress; don’t leave it until the last minute!
  - Look at `example_program_{ll|ht}.c` for typical usage of lists and hash tables
Adminstrivia

❖ Gitlab repo usage
  ▪ Commit things regularly
    • Newly completed units of work / milestones / project parts
    • End-of-day when wrapping up on one computer so you can later pull changes to a different machine
    • And: for this remote quarter, before “visiting” office hours to make it easier for you and TA to browse code
    • etc.
  ▪ Provides backup: protection against lost files and ability to go back in time to retrieve old versions before they got messed up 😊
  ▪ There shouldn’t be one massive commit the day hw is due
  ▪ But: use it properly
    • Don’t push .o and executable files or other build products
      – Clutter, makes it harder to do clean rebuilds, not portable, etc.
    • Don’t use git as a file transfer program (don’t edit on one machine, commit/push/pull to another, compile, and repeat every few minutes)
Discussion Board Tips

❖ When you post a new message or question, try to drop it into the correct category and use a descriptive title
  ▪ Help others discover or find previous posts related to their questions!

❖ Consider whether your question/post really should be private.
  ▪ If others students can benefit from it, you may want to make the post public (but can still be anonymous)
  ▪ Logistical problems specific to you are probably better for private posts.
Lecture Outline

❖ Heap-allocated Memory
  ▪ malloc() and free()
  ▪ Memory leaks

❖ structs and typedef
Memory Allocation So Far

❖ So far, we have seen two kinds of memory allocation:

- **counter** is *statically*-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

- **a, x, y** are *automatically*-allocated
  - Allocated when function is called
  - Deallocated when function returns
Dynamic Allocation

❖ What we want is dynamically-allocated memory
  ▪ Your program explicitly requests a new block of memory
    • The language allocates it at runtime, perhaps with help from OS
  ▪ Dynamically-allocated memory persists until either:
    • Your code explicitly deallocated it (manual memory management)
    • A garbage collector collects it (automatic memory management)

❖ C requires you to manually manage memory
  ▪ Gives you more control, but causes headaches
Why Dynamic Allocation?

❖ Situations where static and automatic allocation aren’t sufficient:
  - We need memory that persists across multiple function calls but not for the whole lifetime of the program
  - We need more memory than can fit on the stack
  - We need memory whose size is not known in advance

```c
// this is pseudo-C code
def char* ReadFile(char* filename) {
    int size = GetFileSize(filename);
    char* buffer = AllocateMem(size);
    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```
Aside: NULL

- NULL is a memory location that is guaranteed to be invalid
  - In C on Linux, NULL is 0x0 and an attempt to dereference NULL causes a segmentation fault
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It’s better to cause a segfault than to allow the corruption of memory!

```c
int main(int argc, char** argv) {
    int* p = NULL;
    *p = 1;  // causes a segmentation fault
    return EXIT_SUCCESS;
}
```
malloc()

❖ General usage: \[ \text{var} = (\text{type}*) \text{malloc}(\text{size in bytes}) \]

❖ malloc allocates a block of memory of the requested size
  ▪ Returns a pointer to the first byte of that memory
    • And returns NULL if the memory allocation failed! // Check this!
  ▪ You should assume that the memory initially contains garbage
  ▪ You’ll typically use sizeof to calculate the size you need

```c
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
    return errcode;
}
... // do stuff with arr
```
**calloc()**

- **General usage:**
  
  ```
  var = (type*) calloc(num, bytes per element)
  ```

- **Like malloc, but also zeros out the block of memory**
  - Helpful when zero-initialization wanted (but don’t use it to mask bugs – fix those)
  - Slightly slower; but useful for non-performance-critical code or if you really are planning to zero out the new block of memory

- **malloc and calloc are found in stdlib.h**

```c
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
    return errcode;
}
...  // do stuff with arr
```
free()

❖ Usage: `free(pointer);`

❖ Deallocates the memory pointed-to by the pointer
  ▪ Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by `malloc` or `calloc`)
  ▪ Freed memory becomes eligible for future allocation
  ▪ `free(NULL);` does nothing.
  ▪ The bits in the pointer are *not changed* by calling free

   • Defensive programming: can set pointer to `NULL` after freeing it

```c
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
    return errcode;
...
    // do stuff with arr
free(arr);
arr = NULL;  // OPTIONAL
```
The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap; `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks
    - Lab 5 from 351!
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.

OS kernel [protected]

Stack

<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>nums</td>
</tr>
<tr>
<td>ncopy</td>
</tr>
</tbody>
</table>

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>  

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.

OS kernel [protected]

Stack

main

nums 1 2 3 4
ncopy

copy

a
i
size 4

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.

OS kernel [protected]

Stack

main

nums 1 2 3 4

ncopy

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
# Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

OS kernel [protected]

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>nums</td>
</tr>
<tr>
<td>main</td>
</tr>
<tr>
<td>ncopy</td>
</tr>
<tr>
<td>free</td>
</tr>
</tbody>
</table>

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (main, copy)

Note: Arrow points to *next* instruction.
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return 0;
}
```

Note: Arrow points to next instruction.
Which line below is first \textit{guaranteed} to cause an error?

A. Line 1

B. Line 4

C. Line 6

D. Line 7

E. We’re lost...
Memory Corruption

❖ There are all sorts of ways to corrupt memory in C

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;  // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3;  // Ok, but if we use c, problem
    free(&a[0])); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

memcorrupt.c
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5; // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3; // Ok, but if we use c, problem
    free(&a[0]); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer

    // any many more!
    return 0;
}

Note: Arrow points to next instruction.

stack:

main

a

b

c

heap:

memcorrupt.c
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5; // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3; // Ok, but if we use c, problem
    free(&a[0]); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;  // assigns past the end of an array
    b[0] += 2;  // assumes malloc zeros out memory
    c = b+3;  // Ok, but if we use c, problem
    free(&a[0]);  // free something not malloc'ed
    free(b);
    free(b);  // double-free the same block
    b[0] = 5;  // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;
    a[2] = 5; // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3; // Ok, but if we use c, problem
    free(&a[0]); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer
    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.

```c
memcorrupt.c
```
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5; // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3; // Ok, but if we use c, problem
    free(&(a[0])); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;        // assigns past the end of an array
    b[0] += 2;       // assumes malloc zeros out memory
    c = b+3;         // Ok, but if we use c, problem
    free(&a[0]));    // free something not malloc'ed
    free(b);
    free(b);         // double-free the same block
    b[0] = 5;        // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;  // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;  // Ok, but if we use c, problem
  free(&a[0]);  // free something not malloc'ed
  free(b);
  free(b);  // double-free the same block
  b[0] = 5;  // use a freed (dangling) pointer

  // any many more!
  return 0;
}

Note: Arrow points to next instruction.
Memory Corruption - What Happens?

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;  // assigns past the end of an array
    b[0] += 2;  // assumes malloc zeros out memory
    c = b+3;  // Ok, but if we use c, problem
    free(&a[0]);  // free something not malloc'ed
    free(b);
    free(b);  // double-free the same block
    b[0] = 5;  // use a freed (dangling) pointer

    // any many more!
    return 0;
}
```

Note: Arrow points to next instruction.
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5; // assigns past the end of an array
    b[0] += 2; // assumes malloc zeros out memory
    c = b+3; // Ok, but if we use c, problem
    free(&a[0])); // free something not malloc'ed
    free(b);
    free(b); // double-free the same block
    b[0] = 5; // use a freed (dangling) pointer

    // any many more!
    return 0;
}
Memory Leak

❖ A memory leak occurs when code fails to deallocate dynamically-allocated memory that is no longer used
  ▪ e.g. forget to free malloc-ed block, lose/change pointer to malloc-ed block

❖ What happens: program’s VM footprint will keep growing
  ▪ This might be OK for short-lived program, since all memory is deallocated when program ends
  ▪ Usually has bad repercussions for long-lived programs
    • Might slow down over time (e.g. lead to VM thrashing)
    • Might exhaust all available memory and crash
    • Other programs might get starved of memory
Lecture Outline

❖ Heap-allocated Memory
  ▪ `malloc()` and `free()`
  ▪ Memory leaks

❖ structs and typedef
Structured Data

- **A `struct`** is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Act similarly to primitive variables
  - A struct `tagname` is a *tag*; not a full first-class type name

- **Generic declaration:**

```c
struct tagname {
    type1 name1;
    ...
    typeN nameN;
};
```

```c
// the following defines a new // structured datatype called // a "struct Point"
struct Point {
    float x, y;
};

// declare and initialize a // struct Point variable
struct Point origin = {0.0, 0.0};
```
Using structs

❖ Use “.” to refer to a field in a struct
❖ Use “->” to refer to a field from a struct pointer
  ▪ Dereferences pointer first, then accesses field

```
struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 0.0};  // p1 is stack allocated
    struct Point* p1_ptr = &p1;

    p1.x = 1.0;
    p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
    return 0;
}
```

simplestruct.c
Copy by Assignment

❖ You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```c
#include <stdio.h>

struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 2.0};
    struct Point p2 = {4.0, 6.0};

    printf("p1: \(%.2f, %.2f\)  p2: \(%.2f, %.2f\)\n", p1.x, p1.y, p2.x, p2.y);
    p2 = p1;
    printf("p1: \(%.2f, %.2f\)  p2: \(%.2f, %.2f\)\n", p1.x, p1.y, p2.x, p2.y);
    return 0;
}
```

`structassign.c`
typedef

- Generic format: `typedef type name;`
- Allows you to define new data type `names/synonyms`
  - Both `type` and `name` are usable and refer to the same type
  - Be careful with pointers – `*` before `name` is part of `type`!

```c
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
    superlong x;
    superlong y;
} Point, *PointPtr;  // similar syntax to "int n, *p;"

Point origin = {0, 0};
```
Dynamically-allocated Structs

- You can `malloc` and `free` structs, just like other data type
  - `sizeof` is particularly helpful here

```c
// a complex number is a + bi
typedef struct complex_st {
    double real;  // real component
    double imag;  // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
    Complex* retval = (Complex*) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```

`complexstruct.c`
Structs as Arguments

- Structs are passed by value, like everything else in C
  - Entire struct is copied – where?
  - To manipulate a struct argument, pass a pointer instead

```c
typedef struct point_st {
    int x, y;
} Point, *PointPtr;

void DoubleXBroken(Point p) { p.x *= 2; }
void DoubleXWorks(PointPtr p) { p->x *= 2; }

int main(int argc, char** argv) {
    Point a = {1,1};
    DoubleXBroken(a);
    printf("(%d,%d)\n", a.x, a.y); // prints: ( , )
    DoubleXWorks(&a);
    printf("(%d,%d)\n", a.x, a.y); // prints: ( , )
    return 0;
}
```
Returning Structs

❖ Exact method of return depends on calling conventions
  ▪ Often in %rax and %rdx for small structs
  ▪ Often returned in memory for larger structs

```c
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex, *ComplexPtr;

Complex MultiplyComplex(Complex x, Complex y) {
    Complex retval;

    retval.real = (x.real * y.real) - (x.imag * y.imag);
    retval.imag = (x.imag * y.real) - (x.real * y.imag);
    return retval; // returns a copy of retval
}
```

complexstruct.c
Pass Copy of Struct or Pointer?

- **Value passed**: passing a pointer is cheaper and takes less space unless struct is small

- **Field access**: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize

- For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers
Extra Exercise #1

❖ Write a program that defines:

▪ A new structured type Point
  • Represent it with floats for the x and y coordinates

▪ A new structured type Rectangle
  • Assume its sides are parallel to the x-axis and y-axis
  • Represent it with the bottom-left and top-right Points

▪ A function that computes and returns the area of a Rectangle

▪ A function that tests whether a Point is inside of a Rectangle
Extra Exercise #2

- Implement `AllocSet()` and `FreeSet()`
  - `AllocSet()` needs to use `malloc` twice: once to allocate a new `ComplexSet` and once to allocate the “points” field inside it
  - `FreeSet()` needs to use `free` twice

```c
typedef struct complex_st {
    double real;  // real component
    double imag;  // imaginary component
} Complex;

typedef struct complex_set_st {
    double num_points_in_set;
    Complex* points;   // an array of Complex
} ComplexSet;

ComplexSet* AllocSet(Complex c_arr[], int size);
void FreeSet(ComplexSet* set);
```