The Heap and Structs CSE 333 Spring 2020

Instructor: Hal Perkins

Teaching Assistants:

Ramya Challa Greg Guo Travis McGaha Cosmo Wang Haoran Yu

Mengqui Chen Zachary Keyes Arjun Singh Yifan Xu Velocity Yu John Depaszthory CJ Lin Guramrit Singh Robin Yang

Administrivia

- hw0 due tonight before 11:00 pm (and 0 seconds)
 - If your clock says 11:01, then it's late!
 - You really, *really* don't want to use late days for hw0
 - 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
- Exercise 3 out today and due Wednesday morning

Yet More Administrivia (sorry)

- Exercise grading Gradescope abuse
 - Grading score is an overall evaluation: 3/2/1/0 = superior / good / marginal / not sufficient for credit
 - 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"

Administrivia

- hw1 due a week from Thursday
 - You *may not* modify interfaces (. h files)
 - But do read the interfaces while you're writing code(!)
 - Suggestion: look at example_program_{ll|ht}.c for typical usage of lists and hash tables
 - Suggestion: have more fun, less anxiety: pace yourself and make steady progress; don't leave it until the last minute!
- Remember: the only supported systems for the class are the Allen School Linux machines (workstations, attus, home VM).
 You should be working on those systems and the projects you build *must* work there.
 - We do not have the cycles to try to support other Unix-like things or chase bugs due to configuration or software differences (including file transfers to/from Windows systems and editing in native Windows environments)

Administrivia

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 "vistiing" 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 ⁽²⁾
- 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

- Please try to post text, not pictures of text
 - Tiny dark color text on a black background is hard to read
 - But use pictures if, in fact, the picture is the useful info
- 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!

Lecture Outline

- * Heap-allocated Memory
 - malloc() and free()
 - Memory leaks
- $\boldsymbol{\ast}$ structs and typedef

Memory Allocation So Far

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

```
int counter = 0; // global var
int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

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

```
int foo(int a) {
    int x = a + 1;    // local var
    return x;
}
int main(int argc, char** argv) {
    int y = foo(10);    // local var
    printf("y = %d\n",y);
    return 0;
}
```

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

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

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

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 (<u>manual memory management</u>)
 - A garbage collector collects it (*automatic memory management*)
- C requires you to manually manage memory
 - Gives you more control, but causes headaches

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!

```
segfault.c
int main(int argc, char** argv) {
    int* p = NULL;
    *p = 1; // causes a segmentation fault
    return 0;
}
```

malloc()

- General usage: (var = (type*) malloc(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!
 - You should assume that the memory initially contains garbage
 - You'll typically use sizeof to calculate the size you need

```
// 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

```
// 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
 - The bits in the pointer are *not changed* by calling free
 - Defensive programming: can set pointer to NULL after freeing it

```
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!



arraycopy.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++)</pre>
    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;
```

<u>Note</u>: Arrow points to *next* instruction.



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



```
#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++)</pre>
    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;
```



Memory Corruption

There are all sorts of ways to corrupt memory in 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; // assign past the end of an array
 b[0] += 2; // assume malloc zeros out memory
 c = b+3; // mess up your pointer arithmetic
 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?



memcorrupt.c

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:

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

```
// 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!

```
#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: {%f,%f} p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
  p2 = p1;
  printf("p1: {%f,%f} p2: {%f,%f}\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!

```
// 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

```
// 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

```
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

```
// 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 stucts (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

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