CSE 333
Lecture 4 - malloc, free, struct, typedef

Administrivia

HW1 is due on Friday, January 25th, 11:15am
- see course overview web page for the late policy

Be sure to check out the course discussion board
- there have been a bunch of great questions and answers about C, HW1, etc.
- I recommend the “mail me whenever a new posting appears” setting
- If possible post questions here, rather than emailing me or the TAs, so that everybody benefits from answers
Administrivia

We *highly* recommend doing the exercises that are at the end of each lecture

- also, Google for “C pointer exercises” and do as many as you can get your hands on
- you MUST master pointers quickly, or you’ll have problems for the rest of the course

Double pointers

what’s the difference between a (char *) and a (char **)?

```c
int main(int argc, char **argv) {
    char hi[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
    char *p, **dp;
    p = &hi[0];
    dp = &p;

    printf("%c %c\n", *p, **dp);
    printf("%p %p %p\n", p, *dp, hi);
    p += 1;
    printf("%c %c\n", *p, **dp);
    printf("%p %p %p\n", p, *dp, hi);
    *dp += 2;
    printf("%c %c\n", *p, **dp);
    printf("%p %p %p\n", p, *dp, hi);
    return 0;
}
```

Exercise 0: draw / update the box-and-arrow diagram for this program as it executes
Double pointers and main()

$ ./a.out 1.0 two three

```
argc = 4
argv = 1.0
   t
   w
   o
   0
   h
   r
   e
   e
   0
```

```
int main( int argc, char **argv) { ... }

or

int main( int argc, char *argv[]) { ... }
```

Today’s goals:

- understand heap-allocated memory
  - malloc(), free()
  - memory leaks
- quick intro to structs and typedef
Memory allocation

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 on entry to block
  - deallocated on exit

---

We need more flexibility

Sometimes we want to allocate memory that:

- persists across multiple function calls but for less than the lifetime of the program
- is too big to fit on the stack
- is allocated and returned by a function and its size is not known in advance to the caller

```// (this is pseudo-C-code)
char *ReadFile(char *filename) {
    int size = FileSize(filename);
    char *buffer = AllocateMemory(size);
    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```
But, you already knew that…

In Java:

```java
PersonRecord p = new PersonRecord();
```

The Object is created when you execute that statement.

What did `new` do?

- Allocate memory to hold instance variables
- Invoke the `PersonRecord` constructor to initialize it

How long does the object live?

- Until your program can no longer reference it.
  (Automatic garbage collection.)

Dynamic memory allocation

Your program explicitly requests a new block of memory:

- the language runtime allocates it, perhaps with help from OS

Dynamically allocated memory persists until:

- your code explicitly deallocates it  [manual memory management]
- a garbage collector collects it  [automatic memory management]

C requires you to manually manage memory

- Why?
C and `malloc`

\[ \text{variable} = (\text{type} *) \text{malloc(size in bytes)}; \]

`malloc` allocates a block of memory of the given size

- returns a pointer to the first byte of that memory
  - `malloc` returns NULL if the memory could not be allocated
- you should assume 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;
arr[0] = 5.1; // etc.
```

C and `calloc`

\[ \text{variable} = (\text{type} *) \text{calloc(#items, sizeof(1 item))}; \]

Just like `malloc`, but also zeroes out the block of memory

- helpful for shaking out bugs
- slightly slower; preferred for non-performance-critical code
- `malloc` and `calloc` are found in `stdlib.h`

```c
// allocate a 10 long-int array
long *arr = (long *) calloc(10, sizeof(long));
if (arr == NULL)
    return errcode;
arr[0] = 5L; // etc.
```
Deallocation

`free(pointer);`

Releases 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()`
- after `free()` ‘ing a block of memory, that block of memory might be returned in some future `malloc()` / `calloc()`
- it’s good form to set a pointer to NULL after freeing it

```c
long *arr = (long *) calloc(sizeof(long),10);
if (arr == NULL)
    return errcode;
// .. do something ..
free(arr);
arr = NULL; // a useful convention
```

Heap

The heap (aka “free store”)

- is a large pool of unused memory that is used for dynamically allocated data
- `malloc` allocates chunks of data in the heap, `free` deallocates data
- `malloc` maintains bookkeeping data in the heap to track allocated blocks
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

arraycopy.c

OS kernel [protected]

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>argc, argv</td>
</tr>
<tr>
<td>nums</td>
</tr>
<tr>
<td>ncopy</td>
</tr>
</tbody>
</table>

heap (malloc/free)

read/write segment
globals

read-only segment
(main, f, g)
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

These are slightly modified versions of slides prepared by Steve Gribble
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
  int *ncopy = copy(nums, 4);
  // ... do stuff ...
  free(ncopy);
  return 0;
}
```

Heap + stack

```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 nums[4] = {2, 4, 6, 8};
  int *ncopy = copy(nums, 4);
  // ... do stuff ...
  free(ncopy);
  return 0;
}
```
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

arraycopy.c

---

Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

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++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2,4,6,8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

arraycopy.c
Heap + stack

```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 nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```

NULL

**NULL:** a guaranteed-to-be-invalid memory location

- an attempt to dereference NULL causes a segmentation fault

In C on Linux:

- NULL is 0x00000000

That’s why you should NULL a pointer after you have free()’d it

- it’s better to have a segfault than to corrupt memory!

```c
#include <stdio.h>

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

    a[2] = 5;    // assign past the end of an array
    a[0] += 2;   // assume malloc zeroes 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 free()'d pointer

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

Memory leak

A memory leak happens when code fails to deallocate dynamically allocated memory that it can no longer reach

```c
// assume we have access to functions FileLen,
// ReadFileIntoBuffer, and NumWordsInString.

int NumWordsInFile(char *filename) {
    char *filebuf = (char *) malloc(FileLen(filename)+1);
    if (filebuf == NULL)
        return -1;

    ReadFileIntoBuffer(filename, filebuf);

    // leak! we never free(filebuf)
    return NumWordsInString(filebuf);
}
```
Implications of a leak?

Your program’s virtual memory footprint will keep growing
- for short-lived programs, this might be OK
- for long-lived programs, this usually has bad repercussions
  - might slow down over time (VM thrashing – see cse451)
  - potential “DoS attack” if a server leaks memory
  - might exhaust all available memory and crash
  - other programs might get starved of memory
- in some cases, you might prefer to leak memory than to corrupt memory with a buggy free()

Structured data

struct typename {
    type name;
    type name;
    ...
    type name;
};

// The following defines a new structured data type with name “struct Point”
struct Point {
    float x, y;
};
struct Point origin = (0.0, 0.0);

struct: a C type that contains a set of fields
- similar to a Java class, but without methods / constructors
- instances can be allocated on the stack or heap
- useful for defining new structured types of data
Using structs

Use "." to refer to fields in a struct

Use "->" to refer to fields through a pointer to a struct

```c
struct Point {   // how much space do these lines allocate?    float x, y;    }

int main(int argc, char **argv) {    int i = 1;    struct Point p1 = {0.0, 0.0}; // p1 is stack allocated    struct Point *pt_ptr = &p1;    p1.x = 1.0;    pt_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: \{%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

These are slightly modified versions of slides prepared by Steve Gribble
typedef
type name;

Allows you to define a new type whose name is *name*
- especially useful when dealing with structs

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

// make "Point" be a synonym for "struct point_st { ... }"
typedef struct point_st { 
    superlong x;
    superlong y;
} Point;
Point origin = {0, 0};
```

---

**structs as arguments**

```c
// Point is a (struct point_st)
// PointPtr is a (struct point_st *)
typedef struct point_st { 
    int x, y;
} Point, *PointPtr, **PointPtrPtr;

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);
    DoubleXWorks(a);
    printf("(%d,%d)\n", a.x, a.y);
    return 0;
}
```

---

These are slightly modified versions of slides prepared by Steve Gribble
You can return structs

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

Complex AddComplex(Complex x, Complex y) {
    Complex retval;
    retval.real = x.real + y.real;
    retval.imag = x.imag + y.imag;
    return retval; // returns a copy of retval
}

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;
}
```

Dynamically allocated structs

You can malloc and free structs, as with other types
- `sizeof` is particularly helpful here

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

ComplexPtr AllocComplex(double real, double imag) {
    Complex *retval = (Complex *) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```
Exercise 1

Write and test a program that defines:

- a new structured type `Point`
  - represent it with floats for the x, y coordinate
- 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/returns the area of a `Rectangle`
- a function that tests whether a `Point` is in a `Rectangle`

Exercise 2

Implement `AllocSet( )`, `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
See you on Friday!