Pointers, Pointers, Pointers!

Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers, Pointers!

Colin Gordon
csgordon@cs.washington.edu

University of Washington

CSE333 Section 2, 4/7/11

Today's Topics

- Pointers
- Pointers
- Pointers
- Homework 1

Pointers!

- Pointers to basic types
- Pointers to arrays
- Pointers to structs
- Pointers in structs
- Function pointers
- Crazy Pointers

Basic Pointers

```
int x = 0;
int *p;
p = &x;
*p = 3;
*p++;
```

Final result: `x = 4`

Array Pointers

```
int a[3] = {0};
int *p1, *p2;
p1 = a;
p2 = &a;
printf("p1:%x    p2:%x\n", p1, p2);
p1 = &a[1];
p2++;
printf("p1:%x    p2:%x\n", p1, p2);
```

Should print two lines where each pointer is equal.

Struct Pointers

```
typedef struct point {
    float x, y;
} Point, *PointPtr;
...
float *f_ptr;
Point a = {0.0, 0.0};                      // stack allocate a Point
PointPtr a_ptr = &a;                       // a_ptr points to a
PointPtr b_ptr =
    (PointPtr)malloc(sizeof(Point));       // b_ptr points to a heap allocation
if (b_ptr == NULL) { ... }                 // handle failed allocation
a_ptr->x = 1.0;                            // same as a.x = 1.0
a.y = 2.0;                                 // same as a_ptr->y = 2.0
(*b_ptr)=a;                                // copy assignment of a into the heap
f_ptr = &b_ptr->y;                         // f_ptr points to y field of b_ptr
*f_ptr = 5.0;                              // same as b_ptr->y = 5.0
free(b_ptr);
```

Structs with Structs

Accessing nested structures:

```
typedef struct line {
    Point p1, p2;
} Line, *LinePtr;
...
Line l;
l.p1.x = 0.0;
l.p1.y = 0.0;
l.p2.x = 1.0;
l.p2.y = 1.0;
```

Structs with Pointers

The same structure, with pointers:

```
typedef struct line {
    Point *p1, *p2;
} Line, *LinePtr;
...
Line *l_ptr = (LinePtr)malloc(sizeof(Line));
if (l_ptr == NULL) { ... }
l_ptr->p1 = (PointPtr)malloc(sizeof(Point));
if (l_ptr->p1 == NULL) { ... /* What happens to l_ptr? */}
l_ptr->p2 = (PointPtr)malloc(sizeof(Point));
if (l_ptr->p2 == NULL) { ... }
free(l_ptr->p1);
free(l_ptr->p2);
free(l_ptr);
```

Remember to free things if you fail after allocating.

Recursive Structs (First Try)

Coming from Java, you might naturally try this...

```
typedef struct tnode {
    int val;
    TreeNode left;
    TreeNode right;
} TreeNode, *TreeNodePtr;
```

But what is `sizeof(TreeNode)`?

Recursive Structs (Second Try)

But we *always* know the size of a pointer...

```
typedef struct tnode {
    int val;
    struct tnode* left;
    struct tnode* right;
} TreeNode, *TreeNodePtr;
```

Under the hood, this is essentially what Java does when you declare a class whose members point to objects of its own class.

Function Pointers

```
typedef void (*FuncPtr)(int*);
```

Why Name a Function Type?

To pass functions as arguments! Think Java event listeners.

From the homework:

```
typedef void(*PayloadFreeFnPtr)(void *payload);
...
void FreeLinkedList(LinkedList list,
                    PayloadFreeFnPtr payload_free_function) {
    ...
    ... payload_free_function(...) ...
    ...
}
```

Out Arguments

A common pattern when you need to return multiple things.

```
/*
 * Allocate an integer array of n integers in the out argument result.
 * Return true for success, or false if there was not enough memory.
 */
bool allocIntArray(int n, int** result) {
    *result = (int*)malloc(n*sizeof(int));
    return (*result != NULL);
}
...
int main(int argc, char* argv[]) {
    int* arrayOfInts = NULL;
    bool success = false;
    ...
    success = allocIntArray(1024, &arrayOfInts);
    if (!success) {
        printf("Ran out of memory!!!\n");
        return -1;
    }
    arrayOfInts[0] = ...
    ...
}
```

Crazy Pointers

What's this pointer?

`int*`

Answer

A pointer to an `int`

Crazy Pointers

What's this pointer?

`int[]`

Answer

A pointer to an `int`

OR

an int array

Crazy Pointers

What's the difference between these pointers?

`int*` and `int[]`

Answer

Mostly convention; both are pointers to `int`s, by convention `int[]` will often be used when that `int` is the first element of an array, and a function argument (`int x[];` is not a valid local variable).

Crazy Pointers

What's this pointer?

`int**`

Answer

A pointer to a pointer to an `int`

Crazy Pointers

What's this pointer?

`int*[]`

or used to declare a variable,

`int* x[];`

Answer

A pointer to a pointer to an `int`

OR

(by convention) an array of `int` pointers

Crazy Pointers

Given these declarations:

`int x;`

What is the type of this expression?

`&x`

Answer

A pointer to an `int`; `int*`

Crazy Pointers

Given these declarations:

```
typedef struct treenode {
 ...
 struct treenode* left;
 struct treenode* right;
 ...
} TreeNode, *TreeNodePtr;
TreeNode n;
```

What is the type of this expression?

`&n.left`

Answer

A pointer to a pointer to a `TreeNode`; a `TreeNode**` or `TreeNodePtr*`

Crazy Pointers

Given these declarations:

```
int x[1024];
```

What is the type of this expression?

`x`

Answer

A pointer to an `int`; `int*`

Crazy Pointers

Given these declarations:

```
int x[1024] = {0};
```

What is the result of evaluating of this expression?

```
x[1024]
```

Answer

Unknown! This goes off the end of the array.

Crazy Pointers

Given these declarations:

```
int x[1024] = {0};
```

What is the difference between these expressions?

```
*x = 1;
```

and

```
x[0] = 1;
```

Answer

Only syntax; they do the same thing

Crazy Pointers

Given these declarations:

```
typedef void (*FancyFunc)(int*, char*);
void f(int *i, char *c) { ... }
```

What is the type of this expression?

`f`

Answer

Either `void (*)(int*,char*)` or `FancyFunc`

Crazy Pointers

Given these declarations:

```
typedef void (*FancyFunc)(int*, char*);
void f(int *i, char *c) { ... }
FancyFunc func;
FancyFunc *func_ptr
```

Is this valid code? What does it do?

```
func_ptr = &func;
*func_ptr = f
```

Answer

It is valid. It stores the address of `f` in the function pointer local variable `func`.

Homework 1

- Overview
- Hash functions

Overview

To the web!

Hash Tables

- Essentially a key-value map; insert a value for a specific key, and look it up later.
- The key property is that unlike tree maps or dictionary lists, looking up a key in a hash table is *ammortized O(1) time*.
- Covered in detail in 332

Chained Hash Tables

The most common form is called a *chained* hash table.

- An array of *buckets*, each containing (a pointer to) a linked-list (chain) of key-value pairs.
- Once you know which bucket to look in, searching for a key is easy: search through the linked list in that bucket for a pair with the right key, and return the corresponding value.

So how do you know which bucket the key is in?

Hash Functions

- Generally, a *hash function* is a function that reduces some arbitrary amount of data to a small number, like an integer.
- This provides a small data for (estimating) equality of much larger pieces of data.
- They are used in many places: for example, cryptography, file system compression, and *hash tables*
- Different use cases desire different properties of their hash functions; for hash tables

Hash tables use hash functions to map keys to a specific bucket. Clients must chew up whatever value they want as a key into something suitable to hash. The ideal hash function for a hash table will distribute keys to buckets roughly evenly (more about this in 332).

Homework 1's Hash Function

```
uint64_t FNVHash64(unsigned char *buffer, unsigned int len) {
  // This code is adapted from code by Landon Curt Noll
  // and Bonelli Nicola:
  //
  // http://code.google.com/p/nicola-bonelli-repo/
  static const uint64_t FNV1_64_INIT = 0xcbf29ce484222325ULL;
  static const uint64_t FNV_64_PRIME = 0x100000001b3ULL;
  unsigned char *bp = (unsigned char *) buffer;
  unsigned char *be = bp + len;
  uint64_t hval = FNV1_64_INIT;

  /*
   * FNV-1a hash each octet of the buffer
   */
  while (bp < be) {
    /* xor the bottom with the current octet */
    hval ^= (uint64_t) * bp++;
    /* multiply by the 64 bit FNV magic prime mod 2^64 */
    hval *= FNV_64_PRIME;
  }
  /* return our new hash value */
  return hval;
}
```

How To Generate a Key

For integers, we provide a helper function:

```
uint64_t key = FNVHashInt64(100);
```

For more general structures, you can either:

1. Cast the address to an unsigned 64-bit integer and hash that (only gives the same hash for the same exact memory address, rather than semantically equal structures), or
2. Convert the structure's semantically meaningful information to a bunch of bytes, and hash that: `uint64_t key = FNVHash64(point_ptr, sizeof(Point))`
   - Gets much trickier if that structure contains pointers!

These uses of the hash function essentially correspond to Java's `.hashCode()` method.

Key→Bucket

There is also the issue of turning the key into a bucket number, which is also a hash (though much simpler in our case), mapping 64-bit integers to integers in the interval `[0,ht->num\_buckets)`:

```
uint32_t HashKeyToBucketNum(HashTableRecordPtr ht, uint64_t key) {
  return (uint32_t) (key % ((uint64_t) ht->num_buckets) );
}
```