Lecture 18:
Cast, Deques, Const, Static
Plan For The Rest Of The Term (1/2)

• 3D: assigned over break
  – Major project, worth 8%
  – Grade options for partial credit

• No projects after 3D
Plan For The Rest Of The Term (2/2)

- Lecture 18: November 27, more data structures & sorting
- Lecture 19: November 29, finals review & Hank’s unsolicited advice
- Final: Monday Dec 3rd, 8am, in this room
Oversleeping for the final

• From class syllabus:

• “If you miss the Final exam period, you may take the Final afterwards for half-credit. Exceptions for full credit will be granted in appropriate situations. These situations include medical emergencies, etc, and do not include oversleeping, forgetting the day of the final, etc.”
Revising syllabus

• If you run out of late passes, then you may continue to earn half credit on any project up until we take the Final. Once we get to the Final (Monday December 3rd, 2018 at 8am), I will no longer accept homeworks.

• If you run out of late passes, then you may continue to earn half credit on any project up until we take the Final. Once we get to Wednesday December 5th (meaning: Thursday December 6th, 6am) I will no longer accept homeworks.

  – Note: if you misread this statement and think you have until Thursday, there will be no sympathy (don’t ask).
Lab This Week

• Lab this week is important
• Looking at some usage of memory
• The sort of concepts I want you to know for the final
Extra OH this week

• When?
YouTube Video Ideas?

• ???
3D: questions
3D – skip to slide 19 unless following slides are needed for questions
Maps

- A map stores tuples: (key, value)
- You store both key and value
- You retrieve with just the key (and it gives the value)
- If the key is not in the map, then some appropriate value is returned
Python Equivalent: Dictionaries

```python
>>> dict = {
    ...
    "951301000": "Hank Childs",
    ...
    "951302000": "Zayd Hamoudeh",
    ...
    "951303000": "Priya Kudva",
    ...
    "951304000": "Viet Lai"
    ...
}
```

```python
>>> dict["951301000"]
'Hank Childs'
>>> dict["951301001"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '951301001'
```
Functions for the Map Data Structure

• Store
  – Textbook: put

• Fetch
  – Textbook: get

• Other noteworthy functions from textbook:
  – containsKey
  – putUnique
  – isEmpty
  – remove
How To Implement a Map

• Idea #1: Linked Lists
• Idea #2: Hash Tables
How To Implement a Map: Linked Lists

- Make a tuple type (key, value)
- Make a linked list of that tuple type
- Store: insert the tuple into the list
- Fetch: iterate over the linked list, comparing the desired key with the key element of each tuple in the linked list

```c
struct ID_Name_Tuple {
    int UO_id; /* NOTE: Python example used string, not int */
    char *student_name;
};
```
How To Implement a Map: Hash Tables

- Make a hash function to convert keys to indices
- Decide on hash table details (open addressing, chaining)
- Store: hash key, insert tuple into hash table
- Fetch: hash desired key, compare with key in tuple (repeat as appropriate)
3D

- Assigned tomorrow, due Friday November 30th
  - Note: full credit up to Weds December 5th
  - 0% credit as of 6am Thurs Dec 6th
  - (As before, no sympathy for misreading this)
3D

• Implement two types of maps
  – Linked list
  – Hash table

• Analyze performance and write very short report (one paragraph)
New material
Casting

• “casting” converts from one type to another
• Tells compiler that the programmer intends for the type to be changed
• Syntax:
  – type1 var1;
  – type2 var2 = (type2) var1;
• Note: accomplishes this through extra variable declaration (var2)
  – Memory allocated for both var1 and var2
  – var1 will always be type1 ... no way to modify its type
Casting: example 1

```c
int main() {
    double Y = 3.5;
    int X = Y;  // The compiler will automatically cast here.
               // It invokes a non-trivial operation:
               // taking the binary representation of a double
               // precision value (Y) and converting it to
               // the binary representation of an integer.
               // This is real work!
    int X2 = (int) Y;  // this is doing the same cast
                       // operation explicitly
}
```
Casting: example 2

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

int main()
{
    void *p = malloc(12);
    int *i = (int *) p;
    printf("p+1 is %p, i+1 is %p\n", p+1, i+1);
}
```

```
Hanks-iMac:Downloads hank$ gcc cast2.c
Hanks-iMac:Downloads hank$ ./a.out
p+1 is 0x7fe94bc02611, i+1 is 0x7fe94bc02614
```
More on Linked Lists
Linked Lists: Motivation

• Will introduce two data structures: linked lists and doubly linked lists
• Linked lists will be used in 3D – you will implement a “map” using linked lists
• Doubly linked lists will tie into a future data structure (Deques)
A linked list is a linear collection of data elements, in which the linear order is not given by their physical placement in memory. Instead, each node in the list points at the next element in the list. Such a structure enables insertion or removal of elements without reallocation or reorganization of the entire structure, since the data items do not need to be stored contiguously in memory.
Linked List Example/Analogy
Example (from Sventek)

```c
struct node {
    struct node *next;
    char *value;
};

struct node *head;

int search(char *university, struct node *theList) {
    struct node *p;

    for (p = theList; p != NULL; p = p->next)
        if (strcmp(p->value, university) == 0)
            return 1;
    return 0;
}
```
How Do Linked Lists Compare With Arrays?
Comparison With Arrays: Hank’s (Wikipedia’s) Answers

- Can do insertions -- beginning, middle, end
- Indexing much slower
- Extra storage

<table>
<thead>
<tr>
<th></th>
<th>Linked list</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Insert/delete at beginning</td>
<td>$\Theta(1)$</td>
<td>N/A</td>
</tr>
<tr>
<td>Insert/delete at end</td>
<td>$\Theta(1)$ when last element is known; $\Theta(n)$ when last element is unknown</td>
<td>N/A</td>
</tr>
<tr>
<td>Insert/delete in middle</td>
<td>search time + $\Theta(1)^6[7]$</td>
<td>N/A</td>
</tr>
<tr>
<td>Wasted space (average)</td>
<td>$\Theta(n)$</td>
<td>0</td>
</tr>
</tbody>
</table>
Let’s Implement Linked Lists
Doubly Linked List

Linked List

```c
struct node {
    struct node *next;
    char *value;
};
```

Doubly Linked List

```c
typedef struct node {
    struct node *next;
    struct node *prev;
    void *value;
} Node;
```
I am asked such question and I have my own sayings but I am not really sure what to say about cons and pros? Microsoft asked this question to one of its candidates.

Singly linked list allows you to go one way direction. Whereas doubly linked list has two way direction next and previous.

It all comes down to usage. There's a trade off here.

Singly linked list is simpler in terms of implementation, and typically has a smaller memory requirement as it only needs to keep the forward member referencing in place.

Doubly linked list has more efficient iteration, especially if you need to ever iterate in reverse (which is horribly inefficient with a single linked list), and more efficient deletion of specific nodes.
Example of Doubly Linked List

• iPod play list
  – Need reverse to get to previous song
Let’s Implement (NOT) Doubly Linked Lists

• See course webpage, under Lecture 16
Deques
Deque – Double Ended Queue

• New data structure
• Pronounced “deck”
• Queue:
  – $O(1)$ insert
  – $O(1)$ to remove head
  – $O(n)$ to remove tail
• Deque:
  – $O(1)$ insert
  – $O(1)$ to remove head
  – $O(1)$ to remove tail
Deque: key methods

- Insert first
- Insert last
- Remove first
- Remove last

- Real world examples?
From Sventek Book

- Head’s prev is NULL
- Tail’s next is NULL
Example exercise

• Palindrome checker
Misc. Topics
const

• const:
  – is a keyword in C
  – qualifies variables
  – is a mechanism for preventing write access to variables
const example

```c
const int X = 5;
```

The compiler enforces const ... just like public/private access controls
Efficiency

Are any of the three for loops faster than the others? Why or why not?

Answer: NumIterations is slowest … overhead for function calls.

Answer: X is probably faster than Y … compiler can do optimizations where it doesn’t have to do “i < X“ comparisons (loop unrolling)
const arguments to functions

• Functions can use const to guarantee to the calling function that they won’t modify the arguments passed in.

```c
struct Image {
    int width, height;
    unsigned char *buffer;
};

ReadImage(char *filename, Image *);
WriteImage(char *filename, const Image *);
```

read function can’t make the same guarantee

guarantees function won’t modify the Image
const pointers

• Assume a pointer named “P”

• Two distinct ideas:
  – P points to something that is constant
    • P may change, but you cannot modify what it points to via P
  – P must always point to the same thing, but the thing P points to may change.
• Assume a pointer named “P”

• Two distinct ideas:
  – P points to something that is constant
    • P may change, but you cannot modify what it points to via P
  – P must always point to the same thing, but the thing P points to may change.
Idea #1: violates const:

```
*P = 3;
```

OK:

```
int Y = 5; P = &Y;
```

**pointer can change, but you can’t modify the thing it points to**

Idea #2: violates const:

```
int Y = 5; P = &Y;
```

OK:

```
*P = 3;
```

**pointer can’t change, but you can modify the thing it points to**
const pointers

int X = 4;
int *P = &X;

Idea #3:
violates const:
"*P = 3;"
"int Y = 5; P = &Y;"

OK:
none

pointer can’t change, and you can’t modify the thing it points to
**const pointers**

```c
int X = 4;
int *P = &X;
```

**Idea #1:** violates const:

```
*P = 3;
```

**OK:**

```
int Y = 5; P = &Y;
```

Pointer can change, but you can’t modify the thing it points to.
const pointers

int X = 4;
int *P = &X;

Idea #2: violates const:
“int Y = 5; P = &Y;”
OK:
“*P = 3;”

pointer can’t change, but you can modify the thing it points to
const pointers

Idea #3: violates const:
   “*P = 3;”
   “int Y = 5; P = &Y;”

OK:
   none

pointer can’t change, and you can’t modify the thing it points to
const usage

• class Image;
• const Image * ptr;
  – Used a lot: offering the guarantee that the function won’t change the Image ptr points to
• Image * const ptr;
  – Helps with efficiency. Rarely need to worry about this.
• const Image * const ptr;
  – Interview question!!
globals
globals

- You can create global variables that exist outside functions.

```c
#include <stdio.h>
int X = 5;

int main()
{
    printf("X is %d\n", X);
}
```

```bash
fawcett:Documents child$ cat global1.C

fawcett:Documents child$ g++ global1.C
fawcett:Documents child$ ./a.out
X is 5
fawcett:Documents child$ 
```
global variables

- Global variables are initialized before you enter main.

```c
#include <stdio.h>

int initializer()
{
    printf("In initializer\n");
    return 6;
};

int X = initializer();

int main()
{
    printf("In main\n");
    printf("X is %d\n", X);
}
```

```
fawcett:Documents childsl$ cat global2.C
fawcett:Documents childsl$ g++ global2.C
fawcett:Documents childsl$ ./a.out
In initializer
In main
X is 6
```
Storage of global variables...

- global variables are stored in a special part of memory
  - “data segment” (not heap, not stack)
- If you re-use global names, you can have collisions

```c
fawcett:Documents childs$ cat file1.C
int X = 6;

int main()
{
}

fawcett:Documents childs$ g++ -c file1.C
fawcett:Documents childs$ cat file2.C
int X = 7;

int doubler(int Y)
{
    return 2*Y;
}

fawcett:Documents childs$ g++ -c file2.C
fawcett:Documents childs$ g++ file1.o file2.o
ld: duplicate symbol _X in file2.o and file1.o
collect2: ld returned 1 exit status
```
Externs: mechanism for unifying global variables across multiple files

```c
#include <stdio.h>

int count = 0;

int doubler(int Y)
{
    count++;
    return 2*Y;
}

int main()
{
    count++;
    doubler(3);
    printf("count is %d\n", count);
}
```

`extern int count;` is a keyword in C. It tells the compiler that `count` is a global variable defined somewhere else. The types are not necessary, but they are good practice.

```
fawcett:330 childs$ cat file1.C
extern int count;

int doubler(int Y)
{
    count++;
    return 2*Y;
}
fawcett:330 childs$ g++ -c file1.C
fawcett:330 childs$ g++ -c file2.C
fawcett:330 childs$ g++ file1.o file2.o
fawcett:330 childs$ .a.out
count is 2
```
static

- static memory: third kind of memory allocation
  - reserved at compile time
- contrasts with dynamic (heap) and automatic (stack) memory allocations
- accomplished via keyword that modifies variables

There are three distinct usages of statics
static usage #1: persistency within a function

fawcett:330 childds$ cat static1.C
#include <stdio.h>

int fibonacci()
{
    static int last2 = 0;
    static int last1 = 1;
    int rv = last1+last2;
    last2 = last1;
    last1 = rv;
    return rv;
}

int main()
{
    int i;
    for (int i = 0 ; i < 10 ; i++)
        printf("%d\n", fibonacci());
}
static usage #2: making global variables be local to a file

I have no idea why the static keyword is used in this way.

```c
#include <stdio.h>

static int count = 0;

int doubler(int Y)
{
    count++;
    return 2*Y;
}

int main()
{
    count++;
    doubler(3);
    printf("count is %d\n", count);
}
```
Unions

- Union: special data type
  - store many different memory types in one memory location

```c
typedef union
{
    float x;
    int    y;
    char   z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This union has 4 bytes
Unions

128-223-223-72-wireless:330 hank$ cat union.c
#include <stdio.h>

typedef union
{
    float x;
    int    y;
    char   z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5;  /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3;    /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}

128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
Unions Example

typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
typedef enum  
{
    CA,  
    OR,  
    WY
} US_State;

typedef struct  
{
    char *carMake;  
    char *carModel;  
    US_State state;  
    LicensePlate lp;
} CarInfo;

int main()  
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
## Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++ -- ()</td>
<td>Prefix and suffix increment and decrement, function call, array subscripting, structure and union member access, structure and union member access through pointer, compound literal</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>+ - ! ~ (type) * &amp; sizeof _Alignof</td>
<td>Prefix and suffix increment and decrement, unary plus and minus, logical NOT and bitwise NOT, type cast, indirection (dereference), address-of, size-of, alignment requirement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Multiplication, division, and remainder</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Addition and subtraction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise left shift and right shift</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;= &gt;=</td>
<td>For relational operators &lt;= and &gt;= respectively</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td>For relational = and != respectively</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td>Bitwise XOR (exclusive or)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Bitwise OR (inclusive or)</td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 [note 1]</td>
<td>?:</td>
<td>Ternary conditional</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>14</td>
<td>= += -= *= /= %= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
<td>=</td>
<td>Simple assignment, assignment by sum and difference, assignment by product, quotient, and remainder, assignment by bitwise left shift and right shift, assignment by bitwise AND, XOR, OR</td>
</tr>
<tr>
<td>15</td>
<td>,</td>
<td>Comma</td>
<td></td>
</tr>
</tbody>
</table>