CS 137 Part 5
Pointers, Arrays, Malloc, Variable Sized Arrays, Vectors
Exam Wrapper

Silently answer the following questions on paper (for yourself)

• Do you think that the problems on the exam fairly reflected the topics covered in this course?
• What percentage of test preparation was done alone vs with others?
• How much time did you spend
  • Reviewing class notes
  • Reworking old homework problems
  • Working on additional problems
  • Reading a textbook/other sources?
• Estimate how many points you lost on your exam for...
  • Not understand a concept
  • Careless mistakes
  • Not being able to formulate an approach to a problem
  • Other reasons (Explain)
• Based on the above, how will you prepare differently for the final exam? Be specific. Also what can I do to help? (Please relay to class reps).
Pointers

• What if we want functions to change values inside memory that are outside the scope of a function?
• We saw this already when we changed values in an array.
• We can do this with other values as well by using pointers and references.
```c
#include <stdio.h>

int main(void) {
    int i = 6;
    int *p;
    p = &i;
    *p = 10;
    // p now points to 10
    printf("%d \n", i);
    int *q;
    q = p;
    *q = 17;
    printf("%d \n", i);
    int a[] = {1, 2, 3};
    return 0;
}
```
Summary

• `int *p` is a pointer to an integer (read right to left).
• `&p` is the address of operator (it returns where `p` is in memory [for any data type]; this is different than the value that is stored there!).
• `*p` is the dereferencing operator - it will return the value stored in where `p` points to and can be used to modify the argument.
Example

Write a function that swaps two integers in memory
Concrete Example

```c
#include <stdio.h>

void swap(int *p, int *q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main (void) {
    int i = 0; j = 2;
    swap(&i, &j);  // references
    printf("%d %d\n", i, j);
}
```
• Turns out in C, you can swap two integers in just one line!
  \[(x \ ^= y), \ (y \ ^= x), \ (x \ ^= y)\];
• Denote XOR using \(\oplus\).
• Trace this with \(x_0\) and \(y_0\) the starting values:
  • Step 1: \(x\) becomes \(x_0 \oplus y_0\)
  • Step 2: \(y\) becomes \(y_0 \oplus (x_0 \oplus y_0) = x_0\).
  • Step 3: \(x\) becomes \((x_0 \oplus y_0) \oplus x_0 = y_0\).
Example

Write a function that returns a pointer to the largest element in a given array.
In the previous code, we used `a + m` where `a` was a pointer and `m` was an integer.

Here, we’ve once again overloaded the `+` operator.

This is an example of **pointer arithmetic**

Supported operations:
- Add/subtract an integer to/from a pointer
- Subtract one pointer from another (so long as they are the same type)

We can also use comparison operators like `<`, `>`, `<=`, `>=`, `==`, `!=

Let’s see some examples
Example

Reminder: Draw picture.

```c
#include <stdio.h>
int main(void) {
    int a[8] = {2, 3, 4, 5, 6, 7, 8, 9};
    int *p, *q, i;

    p = &(a[2]);  // p points to a[2]
    q = p + 3;   // q points to a[5]
    p += 4;      // p points to a[6]
    q = q - 2;   // q points to a[3]
    i = q - p;   // i = 3 - 6 = -3
    i = p - q;   // i = 6 - 3 = 3
    if (p <= q) printf("less\n");
    else printf("more\n");  // printed
    return 0;
}
```
Caveat

• Warning - Two dimensional stack allocated arrays remember are just glorified one dimensional arrays.

• So when doing pointer arithmetic with two dimensional arrays, remember to just treat it as a row major array and you will be fine.

• Let’s revisit summing an array and finding the largest using pointer arithmetic.
int sum (int a[], int n) {
    int total = 0;
    for (int *p = a; p < a + n; p++)
        total += *p;
    return total;
}
int sum (int a[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += *(a + i);
    return total;
}
Largest

```c
int * largest ( int a[], int n) {
    int *m = a;
    for ( int *p = a +1; p<a+n; p ++)
    {
        if (*p > *m) m=p;
    }
    return m;
}
```
int *largest(int a[], int n) {
    int *m = a;
    for(int *p = a+1; p<a+n; p++){
        if (*p > *m) m=p;
    }
    return m;
}
#include <stdio.h>

int main(void) {
    int a[8] = {9, 4, 5, 999, 2, 4, 3, 0, 5};
    int size = sizeof(a)/sizeof(a[0]);
    printf("%d\n", sum(a, size));
    printf("%d\n", *largest(a, size));
    return 0;
}
Challenge

Determine what the following code prints. Assume \( x \) is at memory address 100 and that \texttt{int} \ has size 4.

```c
#include <stdio.h>
int main(void) {
    int x[5];
    printf("%p\n", x);
    printf("%p\n", x + 1);
    printf("%p\n", &x);
    printf("%p\n", &x + 1);
}
```
Challenge

Determine what the following code prints. Assume \texttt{x} is at memory address 100 and that \texttt{int} has size 4.

\begin{verbatim}
#include <stdio.h>
int main(void) {
    int x[5];
    printf("%p\n", x);       // 100
    printf("%p\n", x + 1);   // 104
    printf("%p\n", &x);     // 100 (x == &x)
    printf("%p\n", &x + 1);
    // 120 (int (*x)[5]+1) mem addy of array
    // then added 1 to entire length.
}
\end{verbatim}
**Final Pointer Arithmetic Comment**

The * operator and ++ operator can be combined:

- *p++ is the same as *(p++) (Use *p first then increment pointer).
- (*p)++ (Use *p first then increment *p).
- +++p or *(++p) (Increment p first then use *p after increment).
- +++p or ++(*p) (Increment *p first then use *p after increment).
#include <stdio.h>
int main(void) {
    int a[4] = {5, 2, 9, 4};
    int sum = 0;
    for (int *p = a; p < a + 4; p++) {
        sum += *p;
    }
    printf("%d", sum);
    return 0;
}

#include <stdio.h>
int main(void) {
    int a[4] = {5, 2, 9, 4};
    int sum = 0;
    int *p = &a[0];
    while (p < &a[4]) {
        sum += *p++;
    }
    printf("%d", sum);
    return 0;
}
Advanced Pointer Topics

• Up to this point, all of our memory usage has been on the stack.
• There are times however where we might want to allocate large chunks of memory or where we might need some dynamically allocated memory.
• This is where the heap and memory allocation concepts will become important.
Slightly More Detailed Code Storage

stack
↓
↑
heap
| constants |
| text |
Stack vs Heap

From openclipart.com

Stack

Heap
Stack vs Heap

Stack
- Scratch space for a thread of execution.
- Each thread gets a stack.
- Elements are ordered (new elements are stacked on older elements).
- Faster since allocating/deallocating memory is very easy.

Heap
- Memory set aside for dynamic allocation.
- Typically only one heap for an entire application.
- Entries might be unordered and chaotic.
- Usually slower since need a lookup table for each element (ie. more bookkeeping).
Commands

To use the following, we need \#include <stdlib.h>.

```c
void *malloc(size_t size);
```

- Allocates block of memory of size number of bytes but doesn’t initialize.
- Returns a pointer to it.
- Returns NULL, the null pointer, if insufficient memory or size==0.

```c
void free(void *)
```

- Frees a memory block that was allocated by user (say using malloc).
- Failure to free memory that you have allocated is called a memory leak.
More on the NULL Pointer

- Since pointers are memory addresses, we need to be able to distinguish from a pointer to something and a pointer to nothing.
- The NULL pointer is how we do this. It can be called by
  - `int *p = NULL;`
  - `int *p = 0;`
  - `int *p = (int *) 0;`
  - `int *p = (void *) 0;`
- The `(void *)` typecast will automatically get converted to the correct type.
- The NULL pointer is in many libraries, including `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<time.h>`, `<wchar.h>` and possibly others.
Create an array of numbers

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

int *numbers(int n);

int main(void) {
    int *q = numbers(100);
    printf("%d\n", q[50]);
    free(q);   // Avoid memory leak
    q = NULL;  // Guards against double deletes
    return 0;
}
```
int *numbers(int n){
    int *p = malloc(n * sizeof(int));
    assert(p); // Verify that malloc succeeded.
    for(int i=0; i<n; i++)
        p[i] = i;
    return p;
}
Other Allocators

Again, we need `<stdlib.h>` to use these.

void* calloc (size_t nmemb, size_t size)

• Clear allocate.
• Allocates \( nmemb \) elements of \( size \) bytes each initialized to 0

void* realloc (void *p, size_t size)

• Resizes a previously allocated block
• May need to create a new block and copy over old block contents.

Typically, `malloc` is used unless you have a good reason to do otherwise.
Pointers to structs

- Let’s revisit our time of day struct example
  ```c
  struct tod {int hour, min};
  ```
- To create a pointer to the structure, we can use:
  ```c
  struct tod *t = malloc (sizeof(struct tod));
  ```
- Now `t` points to the beginning of a struct where the integers `hour` and `min` are located.
- We can modify these values by using `(*t).hour = 18;` or `t->hour = 18;`
- Note: Arrow operator can be overloaded (say in C++) whereas the dot cannot. Brackets are necessary above because dot has precedence. Arrow is left associative (like addition, multiplication, etc.).
Flexible Array Members

- In the time of day example, the sizes of all the elements were fixed.
- What happens if you say want a struct with an array whose size is to be determined later?
- Turns out there are ways to handle this but it must be done very carefully.
- This is valid only in C99 and beyond.
- This technique is called the “struct hack”.
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

struct flex_array{
    int length;
    int a[];  //Note: declared at end
};

• Inside the struct, int a[] has size 0.
• sizeof(struct flex_array) returns 4.
• Note: In <stdlib.h>, there is a data type size_t that should be used when using malloc.
```c
int main(void) {
    size_t array_size = 4;
    struct flex_array *fa = malloc(
        sizeof(struct flex_array)
        + array_size * sizeof(int));
    assert(fa);
    fa->length = array_size;
    for (int i = 0; i < fa->length; i++)
        fa->a[i] = i;
    printf("%d\n", fa->a[3]);
    free(fa);
    fa = NULL;
    return 0;
}
```
Arrays have a fixed size. Is there a way to create an array that expands as more terms are needed?

There is a library in C++ that does this, the `vector` library but not in C.

We’ll actually create a simplified instance of this to demonstrate how it works for a vector of integers.

Idea: Initialize contents to 0 and grow automatically by powers of 2.
#ifndef VECTOR_H
#define VECTOR_H

struct vector;
struct vector *vectorCreate(void);
struct vector *vectorDelete(struct vector *);
void vectorAdd(struct vector *, int val);
void vectorSet(struct vector *, int index, int val);
int vectorGet(struct vector *, int index);
int vectorLength(struct vector *);
#endif

Note: size is the total storage where as length is the actual used storage.
Descriptions (should include in the header file!)

- `struct vector *vectorCreate();` will create a new empty vector.
- `struct vector *vectorDelete(struct vector *v));` deletes the vector *v*. Returns NULL on success. (return NULL to allow for v=vectorDelete(v);
- `void vectorAdd(struct vector *v, int value);` adds value to the end of the vector. Allocates new space as necessary.
- `void vectorSet(struct vector *v, int index, int value);` sets index index to be value.
- `int vectorGet(struct vector *v, int index);` returns element at index index.
- `int vectorLength(struct vector *v);` returns the length of the vector *v.`
#include "vector.h"
#include "assert.h"
#include <stdlib.h>

struct vector{
    int *arr;
    int size, length;
};
struct vector *vectorCreate(void) {
    struct vector *v = malloc(sizeof(struct vector));
    assert(v);
    v->size = 4;
    v->arr = malloc(4*sizeof(int));
    assert(v->a);
    v->length = 0;
    return v;
}

struct vector *vectorDelete(struct vector *v) {
    if (v) {
        free(v->arr);
        free(v);
    }
    return NULL;
}
Vector.c (Continued)

```c
void vectorAdd(struct vector *v, int value) {
    assert(v);
    if (v->length == v->size) {
        int newSize = v->size*2;
        int *newArr = malloc(newSize*sizeof(int));
        for (int i = 0; i < v->size; ++i) {
            newArr[i] = v->arr[i];
        }
        newArr[v->size] = value;
        free(v->arr);
        v->size = newSize;
        v->arr = newArr;
    } else {
        v->arr[v->length] = value;
    }
    ++v->length;
}
```
void vectorSet(struct vector *v, int index, int value) {
    assert(v && index >= 0 && index <= v->length);
    v->arr[index] = value;
}
Vector.c (Continued)

int vectorGet(struct vector *v, int index) {
    assert(v && index >= 0 && index < v->length);
    return v->arr[index];
}

int vectorLength(struct vector *v) {
    assert(v);
    return v->length;
}
int main() {
    struct vector *v = vectorCreate();
    for (int i = 0; i < 20; ++i) {
        vectorAdd(v, i);
    }
    printf("%d\n", vectorLength(v));
    for (int i = 0; i < 20; ++i) {
        printf("v[i]: %d\n", vectorGet(v, i));
    }
    for (int i = 0; i < 20; ++i) {
        vectorSet(v, i, i*i);
        printf("v[i]: %d\n", vectorGet(v, i));
    }
    v = vectorDelete(v);
}
Notice how none of the implementation details were in our header file; only the declarations.

This is a design principle known as **information hiding**.

We do this to hide implementation details from the user, yet keep the user interaction/interface the same.

We can modify the internal code and not affect other people who are using our code externally.

Notice that with this header, `struct vector v` is not possible where as `struct vector *v` is possible (the header doesn’t know the size of the struct since it is implemented in the `.c` file.)