CS 137 Week 5

Pointers, Arrays, Malloc, Variable Sized Arrays, Vectors

October 16th, 2017

Exam Wrapper

Silently answer the following questions on paper (for yourself)

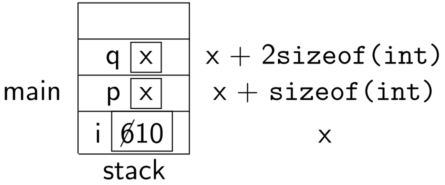
- Do you think that the problems on the exam fairly reflected the topics covered in this course?
- What percentage of test preparation was done alone vs with others?
- How much time did you spend
    - Reviewing class notes
    - Reworking old homework problems
    - Working on additional problems
    - 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.

Example

```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);
   return 0;
}
```



Example

Write a program that swaps two integers in memory

Concrete Example

```
#include <stdio.h>
void swap(int *p, int *q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}
void main () {
    int i = 0; j = 2;
    swap(&i, &j); //references
    printf("%d %d\n", i, j);
}
```

Just For Fun

- 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.

Pointer Arithmetic

- 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.

```
#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 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.

Summing Array

```
int sum (int a[], int n) {
  int total = 0;
  for (int *p = a; p < a + n; p++)
    total += *p;
  return total;
}
```

Summing Array (Alternate)

```
int sum (int a[], int n) {
    int total = 0;
    for (int i = 0; i < n; i ++)
        total += *(a + i);
    return total;
}
```

Largest

Largest

```
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;
}
```

Testing For Previous

```
#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 `int` has size 4.

```
#include <stdio.h>
void 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 x is at memory address 100 and that int has size 4.

```
#include <stdio.h>
void 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.
}
```

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).

Example

```c
#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;
}
```

```c
#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.

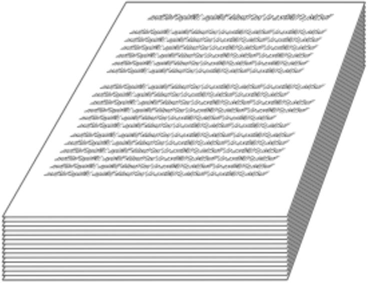
Slightly More Detailed Code Storage

text
constants
heap ↓ ↑ stack

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 on top of 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>`.

`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`.

`void free(void *)`

- Frees a memory block that was allocated by user (say using `malloc`).

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.

Sample

Create an array of numbers

```
#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;
}
```

Code Continued

```
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
- `struct tod int hour,min;`
- To create a pointer to the structure, we can use:

  ```
  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 whereas the dot cannot. Brackets are necessary above because dot has precedence.

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".

Struct Hack Setup

```
#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.

Struct Hack Execution

```
#include <stdio.h>
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;
}
```

Variable Size Array

- 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.
- Idea: Initialize contents to 0 and grow automatically by powers of 2.

Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H
struct vector;
struct vector *vectorCreate();
struct vector *vectorDelete(struct vector *v));
void vectorSet(struct vetor *v, int index,
  int value);
int vectorGet(struct vector *v, int index);
int vectorLength(struct vector *v);
#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 vector and initialize everything to 0.
- `struct vector *vectorDelete(struct vector *v));` deletes the vector *v. Returns NULL on success. (return NULL to allow for v=vectorDelete(v);)
- `void vectorSet(struct vector *v, int index, int value);` sets index `index` to be `value`. This code rescales the vector as necessary.
- `int vectorGet(struct vector *v, int index);` returns element at index `index`.
- `int vectorLength(struct vector *v);` returns the length of the vector *v.

Vector.c

```
#include "vector.h"
struct vector{
  int *a;
  int size, length;
};
```

Vector.c (Continued)

```
struct vector *vectorCreate() {
    vector *v = malloc(sizeof(vector));
    assert(v);
    v->size = 1;
    v->a = malloc(1*sizeof(int));
    assert(v->a);
    v->length = 0;
    return v;
}

struct vector *vectorDelete(struct vector *v) {
    if (v) {
      free(v->a);
      free(v);
    }
    return NULL;
}
```

Vector.c (Continued)

```
void vectorSet(struct vector *v,
int index, int value) {
  assert(v && index >= 0);
  // grow storage if necessary
  if (index >= v->size) {
    do {
      v->size *= 2;
    } while (index >= v->size);
    v->a = realloc(v->a, v->size * sizeof(int));
  }
  //Zero Fill
  while (index >= v->length) {
    v->a[v->length] = 0;
    v->length++;
  }
  v -> a[index] = value;
}
```

Vector.c (Continued)

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

int vectorLength(struct vector *v) {
  assert(v);
  return v->length;
}
```

Test.c

```c
#include <stdio.h>
#include "vector.h"

void main() {
  vector *v = vectorCreate();
  vectorSet(v, 10, 2);
  printf("%d\n", vectorLength(v));
  printf("%d\n", vectorGet(v, 10));
  v = vectorDelete(v);
}
```

Summary Note

- 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.