

CS246

Closing Remarks

May 10

Systems Programming

- 2 Parts
 - C Programming
 - UNIX tools

C Programming

- Why C?
 - Generally it does not hide how it works
 - Arrays are explicitly contiguous blocks of memory
 - Row-major allocation of N dimensional arrays is meaningful
 - Pointers are explicitly just memory locations

The Last C program

- a function
- long arr[cnt];
 - allocates memory on the stack, so it is automatically cleaned up when function completes
 - Contrast to Java – all arrays are from heap

```
void work1(long cnt)
{
    long arr[cnt];
    for (int i = 0; i < cnt; i++) {
        for (int j = 0; j < cnt; j++) {
            arr[i] += j;
        }
    }
}
```

C - 1b

- Mixing array and pointer notations is fine
 - confusing
 - often little benefit

```
void work1b(long cnt)
{
    long arr[cnt];
    for (int i = 0; i < cnt; i++) {
        int *p = (int *)arr;
        for (int j = 0; j < cnt; j++) {
            *(p++) += j;
        }
    }
}
```

C part 2

- Pretty much identical to work1 but with pointers
- A little faster than work1
 - 10%-ish
- malloc allocates from heap
 - Identical to Java
 - No garbage collection in C so
 - “for each malloc there must be a free”
 - reference counting

```
void work2(long cnt) {
    long *arr = malloc(cnt * sizeof(long));
    long *last = arr + cnt;
    for (int i = 0; i < cnt; i++)
    {
        long *p = arr;
        while (1) {
            *(p++) += i;
            if (p==last)
                break;
        }
    }
    free(arr);
}
```

C part 3

- LinkedList style
 - typedef for struct uses double naming so the “struct ll” name is available within the struct.
 - First time through outer loop allocate the linked list
- Every malloc a free

```
typedef struct ll {  
    int val;  
    struct ll *next;  
} ll;
```

```
void work3(long cnt) {  
    ll *head = NULL;  
    for (int i = 0; i < cnt; i++) {  
        if (i==0) {  
            head = malloc(1 * sizeof(ll));  
            ll *this = head;  
            for (int j = 0; j < cnt; j++) {  
                this->next = malloc(1 * sizeof(ll));  
                this->val = i; this = this->next; }  
        } else {  
            ll *this = head;  
            for (int j = 0; j < cnt; j++) {  
                this->val += j; this = this->next; }  
        }  
        while (head != NULL) {  
            ll *nxt = head->next; free(head); head = nxt;  
        }  
    }  
}
```

C final

- Typedef defines a function pointer type
- funarray contains pointers to each work function
- Could have done this but function pointers are more fun
- Still need to check bounds
- Timing!!

```
typedef void(*funs)(long);
```

```
int main(int argc, char const *argv[]) {  
    funs funarray[3];  
    funarray[0] = work1;  
    funarray[1] = work2;  
    funarray[2] = work3;  
    clock_t start, end;  
    start = clock();  
    int funsel = atoi(argv[1]);  
    int funwork = atol(argv[2]);  
    funarray[funsel](funwork);  
    end = clock();  
    float cpu_time_used = ((float) (end - start)) / CL  
    printf(CCOLUMN, funsel, funwork, cpu_time_used );  
    return 0;  
}
```


Now, the System!!

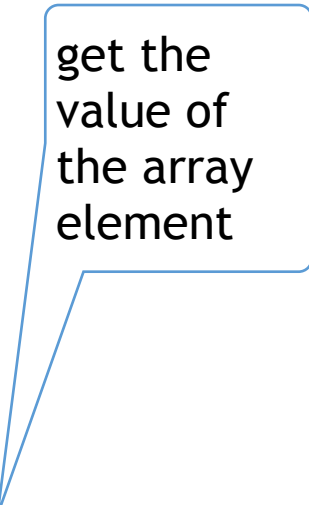
- Use a shell script to collect lots of data
 - Why?
 - in this case could just edit the program to collect the data itself.
 - Often do not have the program source code to edit!
- Can we do better?
 - the `` invoke a “sub shell” which is inefficient
- Also, this spews out a lot of data

```
for ((i=0; i<3; i++))
do
    for ((j=1000; j<40000; j=j*2))
    do
        for ((k=0; k<$1; k++))
        do
            echo `a.out $i $j`
        done
    done
done
```

Beter shell script

- Rather than a sub-shell use `$()` to run in current shell
- `ARR=$(R)` puts result into an array
- bash is limited to integer math so use the “bc” program
 - the “c” is for calculator
 - `-l` gets bc to use floating point
-

```
for ((i=0; i<3; i++))
do
  for ((j=1000; j<20000; j=j*2))
  do
    SUM=0
    for ((k=0; k<$1; k++))
    do
      R=$(a.out $i $j)
      ARR=( $R)
      SUM=$(echo "$SUM + ${ARR[2]}" | bc -l)
    done
    AVG=$(echo "$SUM / $1" | bc -l)
    echo "$i $j AVERAGE over $1: $AVG"
  done
done
```



get the value of the array element

More Systems

- counting “bc” from today ...
- 50 Unix commands
 - cd, ls, pwd, mkdir, rm, cp, mv, ln
 - more, less, head, tail, cat
 - sort, uniq, tr, wc, grep
 - history
 -
- Point: the OS provides a lot of tools to help you so use them creatively to be lazy.
 - A lazy programmer is a good programmer

Shooting yourself in the foot

- APL
 - You shoot yourself in the foot and then spend all day figuring out how to do it in fewer characters.
 - You hear a gunshot and there's a hole in your foot, but you don't remember enough linear algebra to understand what happened.
 - @#&^\$%&%^ foot
- C
 - You shoot yourself in the foot and then nobody else can figure out what you did.

Java

- You write a program to shoot yourself in the foot and put it on the Internet. People all over the world shoot themselves in the foot, and everyone leaves your website hobbling and cursing.
- You amputate your foot at the ankle with a fourteen-pound hacksaw, but you can do it on any platform.
- Lisp
 - You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot...
 - You attempt to shoot yourself in the foot, but the gun jams on a stray parenthesis.
- Linux
 - You shoot yourself in the foot with a Gnu.

Perl

- You separate the bullet from the gun with a hyperoptimized regexp, and then you transport it to your foot using several typeglobs. However, the program fails to run and you can't correct it since you don't understand what the hell it is you've written.
- You stab yourself in the foot repeatedly with an incredibly large and very heavy Swiss Army knife.
- You shoot yourself in the foot and then decide it was so much fun that you invent another six completely different ways to do it.

Python

- You shoot yourself in the foot and then brag for hours about how much more elegantly you did it than if you had been using C or (God forbid) Perl.
-

Obfuscated C

- C has the rep of being obscure and unreadable
- It does not have to be
- IOCCC celebrates that it can!

Lab

- Log in to
 - <http://165.106.10.190/246/login.html>
- Use your UNIX login and the password 246
- From here click on
 - “Link to course evaluation”
 - Evaluate the course
 - Other links are currently dead
- **THIS IS HOW YOU WILL GET THE EXAM**
 - make sure you can log in