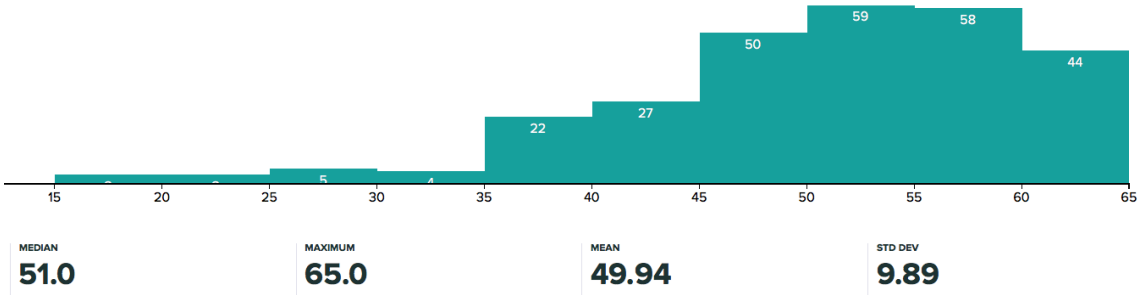
CS107
Lecturer: Chris Gregg Feb 8, 2018

Midterm Examination (KEY)



MEDIAN	MAXIMUM	MEAN	STD DEV
51.0	65.0	49.94	9.89

The figure above is the histogram and class statistics for the midterm. It looks like most of you prepared well for the exam — great job!

The midterm serves as a mid-quarter assessment. Please come by office hours if you have concerns about your progress in the course, or you want to review the material.

Your graded exam is available online at Gradescope. You should receive an email at your sunet@stanford.edu address with information on how to access it.

Grading rubrics:

Our rubrics are designed to apply constructively. Your score starts at zero and we identify in your answer those tasks for which you make a valid attempt and/or accomplish correctly and attach the appropriate rubric items, each of which makes a positive contribution to the score. For tasks that were not attempted or not accomplished correctly, there will be no rubric item applied and there is no change in score for those items.

In the coding questions, care in handling pointers/memory, applying the correct typecast, computing the right offset, and accessing at the appropriate level of indirection were the key skills we were assessing. The bulk of the points are apportioned for those issues, with correspondingly lighter weighting on the mundane tasks (for loops, return values, integer counters, etc.).

Regrades: If you believe there was an error in scoring your exam, please send email to cs107@cs.stanford.edu with an explanation of the concern and ask for our review. The entire exam will be re-evaluated. Regrade requests must be sent by **Mon February 19th.**

1. Bits, bytes, and numbers	9	points
2. C-strings	18	points
3. Pointers and generics	18	points
4. Using qsort	10	points
5. Void * and Function pointers	10	points
Total	**65**	points

Problem 1: Bits, bytes, and numbers (9 points)

Answer parts A, B, and C about the following mystery function:

```
unsigned char mystery(unsigned char n)
{
      n |= n >> 1;
      n |= n >> 2;
      n |= n >> 4;
      n++;
      return (n >> 1);
}
```

A. What does the following code print?

```
printf("%u\n",mystery(17)); // %u prints the integer value
printf("%u\n",mystery(88)); //    for an unsigned char
printf("%u\n",mystery(150));
```

Note: `17 = 0b00010001`, `88 = 0b01011000`, and `150 = 0b10010110`

Output: **(1 point each)**

```
16
64
0
```

B. For what values of `n` does `mystery(n)` return non-zero? **(3 points)**

From 1 to 127

C. When `mystery(n)` returns a non-zero value, what is the general bit pattern of the result? In other words, explain the return value in terms of the argument, `n`. **(3 points)**

The value is the next lowest power of two to `n`. Alternatively: the return value only holds the most significant bit of the original number, and zeros all bits less significant.

Problem 2: C-strings (18 points)

2a) The function

```
char *substr(const char *s, char start, char stop, char result[])
```

populates `result` with the substring that starts at the first instance of `start` and ends at the *next* instance of `stop`. The `result` buffer is guaranteed to be big enough to hold the substring, and the function should properly null-terminate result. If there isn't a substring that meets the criteria, `result` should contain the empty string. The `result` buffer is also returned to the calling function.

Here are some examples:

```
char input_str[] = "Mississippi";
char buffer[sizeof(input_str)];

substr(input_str,'i','p',buffer); // populates buffer with
                                  // "ississip"

substr(input_str,'s','i',buffer); //populates buffer with "ssi"

substr(input_str,'s','s',buffer); // populates buffer with "ss"

substr(input_str,'p','s',buffer); // populates buffer with the
                                  //  empty string.
```

Requirements:

- Your function should not allocate, deallocate, or resize any memory.
- Re-implementing functionality that is available in the standard library will result in loss of credit. For example, your code can not have *any* explicit loops! Instead, call the library functions!

Write the function on the following page.

(11 points)

```
char *substr(const char *s, char start, char stop, char result[])
{

    char s1[0];
    // populate result's first byte with 0
    result[0] = '\0';

    // find the location of the start
    char *start_ptr = strchr(s,start);
    if (!start_ptr) return result;

    // find the location of stop
    char *stop_ptr = strchr(start_ptr+1,stop);
    if (!stop_ptr) return result;

    // copy the bytes
    memcpy(result,start_ptr,stop_ptr-start_ptr+1);

    // null-terminate
    result[stop_ptr-start_ptr+1] = '\0';

    // bad code ahead
    /*
    char *new_buffer = malloc(strlen(result));
    strcpy(new_buffer,result);
    free(result);
    return new_buffer;
    */
    // end of bad code

    return result;
}
```

Problem 2: C-strings (continued)

2b) Your colleague decides that it would make more sense to have a correctly-sized `result` buffer so you don't waste space. So, they suggest adding the following code before returning from the function (after `result` has been populated correctly):

```
// note: caller is responsible for freeing returned pointer
char *new_buffer = malloc(strlen(result));
strcpy(new_buffer,result);
free(result);
return new_buffer;
```

While you are happy that your colleague has left a nice comment about the caller being responsible for freeing the memory, you see that there are two problems in the code. One problem is definitely an error, and the other problem has a big potential to be an error.

What are the two problems? **(3 points each)**

1. `malloc(strlen(result))` allocates 1 fewer byte than we need, as we need an extra byte for the null terminator.
2. `result` may not have been allocated on the heap, so freeing it could cause a runtime error.

When your colleague doesn't believe you, you say, "I can prove it,"
and you run one tool that will point out both errors. What is that tool? **(1 point)**

valgrind

Problem 3: Pointers and generics (18 points)

3a) In class, we discussed a generic stack, with last-in-first-out behavior. For this problem, you will be creating a generic *queue*, which has first-in-first-out behavior. A generic queue has the same `node` as a stack:

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

The `queue` definition is as follows. Note that there is both a front and a back in a queue, and elements are enqueued onto the back of the queue, and dequeued from the front:

```
typedef struct queue {
    int width;
    int nelems;
    node *front;
    node *back;
} queue;
```

The `queue_create` function initializes a queue:

```
queue *queue_create(int width)
{
    // note: caller responsible for freeing queue
    queue *q = malloc(sizeof(queue));
    q->width = width;
    q->nelems = 0;
    q->front = NULL;
    q->back = NULL;
    return q;
}
```

The `queue_enqueue` function works by copying the data into a `node`, and it **does not simply copy the pointer location**. The function looks like this:

```
// data is a pointer to the address where
// queue->width number of bytes will be copied into
// a queue node
void queue_enqueue(queue *q, const void *data)
{
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(q->width);
    memcpy(new_node->data,data,q->width);

    new_node->next = NULL;
```

```
        if (q->front == NULL) {
            q->front = q->back = new_node;
        } else {
            q->back->next = new_node;
            q->back = new_node;
        }
        s->nelems++;
    }
```

Write the queue_dequeue function:

```
/* return value: true if queue has any elements when called,
                 false if queue is empty when called
   addr: pointer to address that can hold queue->width
         bytes from queue node. The data in the node
         at the front of the queue should be copied
         to the address pointed to by addr, and the node
         should be completely removed from the queue.
*/
bool queue_dequeue(queue *q, void *addr)      (10 points)
{
 if (q->nelems == 0) {
        return false;
    }
    node *n = q->front;
    memcpy(addr,n->data,q->width);
    // rewire
    q->front = n->next;
    if (q->front == NULL) {
        q->back = NULL;
    }

    free(n->data);
    free(n);
    q->nelems--;
    return true;
}
```

3b) In assign3, you wrote a *tail* program with a circular queue of a fixed size. Another way to write the program would have been with the generic queue you just created. Fill in the blanks for the following `main` function of a *tail* program. Your program should *not leak any memory*:

```
int main(int argc, char **argv)  (1 point per line)
{
    char buffer[1024]; // maximum size of a line
    int nlines = atoi(argv[1]); // number of lines for tail
    FILE *fp = fopen(argv[2],"r"); // filename to open

    queue *q = queue_create(_sizeof(char *)_); // line 1

    int lines_read = 0;
    char *line;
    while (fgets(buffer,MAX_LINE,fp)) {
        // remove newline
        buffer[strlen(buffer)-1] = '\0';

        // Make a persistent copy of the buffer
        // Store the address of the copy in the queue.

        line = _strdup(buffer)__; // line 2

        queue_enqueue(___q, &line_____); // line 3

        lines_read++;

        if (lines_read > nlines) {

            queue_dequeue(___q, &line____); // line 4


            ___free(line)___; // line 5
        }
    }
    fclose(fp);

    while (queue_dequeue(___q, &line__)) { // line 6

        printf("%s\n",line);

        ___free(line)____; // line 7
    }
    _____free(q)_______; // line 8

    return 0;
}
```

Problem 4: Using qsort (10 points)

Assume the following definition of a date:

```
typedef struct date {
        int month;
        int year;
} date;
```

Dates can be compared by first looking at their year, and if the year is the same, then by comparing the months. For example, `{1,2018}` (January 2018) is less than `{2,2018}` (February 2018), and `{5,2000}` (May 2000) is less than `{5,2018}` (May 2018).

Write a comparison function that can be passed into `qsort` to sort an array of `date`s, as in the following code.

```
int main(int argc, char **argv)
{
    date dates[] = {{1,2000},{2,1999},{3,2000},{2,2018},
                    {12,2005},{8,2007}};
    size_t nelems = sizeof(dates) / sizeof(dates[0]);
    qsort(dates,nelems,sizeof(date),compar_dates);

    // print sorted list
    for (int i=0; i < nelems; i++) {
        date *d = dates + i;
             printf("%d/%d\n",d->month,d->year);
        }
        return 0;
}
```

Hint: remember that a comparison function returns a negative `int` if the first element is less than the second, zero if they are equal, and a positive `int` if the first element is greater than the second element. **(10 points)**

```
int compar_dates(const void *d1, const void *d2)
{
    date *d1_ptr = (date *)d1;
    date *d2_ptr = (date *)d2;
    if (d1_ptr->year < d2_ptr->year) {
        return -1;
    } else if (d1_ptr->year > d2_ptr->year) {
        return 1;
    } else {
        return d1_ptr->month - d2_ptr->month;
    }
}
```

Problem 5: Void * and Function Pointers (10 points)

The "map" function (not to be confused with the map abstract data type) applies another given function to an array of elements. For example, given an array of doubles and a square root function, the map function **applies the square root function to each element in the array**, with the result for each operation replacing the original element value. Here is what that would look like in code:

```
double arr[] = {36,25,144,16};
size_t nelems = sizeof(arr) / sizeof(arr[0]);
map(arr,nelems,sizeof(double),sqrt_fn);
// now arr holds {6, 5, 12, 4};
```

An example of a generic function that would be passed into the map function looks like this:

```
void sqrt_fn(void *d_ptr) {
    double d = *((double *)d_ptr);
    *((double *)d_ptr) = sqrt(d);
}
```

Write the generic `map` function that would perform the above map operation: **(10 points)**

```
void map(void *arr, size_t nelems, int width, void (*map_fn)(void *))
{
    for (size_t i = 0; i < nelems; i++) {
        void *ith = (char *)arr + i * width;
        map_fn(ith);
    }
}
```

Library functions

Reminder of the prototypes for potentially useful functions. You won't use every function listed. You may also use standard C library functions not on this list.

```
size_t strlen(const char *str);
int    strcmp(const char *s, const char *t);
int    strncmp(const char *s, const char *t, size_t n);
char  *strchr(const char *s, int ch);
char  *strstr(const char *haystack, const char *needle);
char  *strcpy(char *dst, const char *src);
char  *strncpy(char *dst, const char *src, size_t n);
char  *strcat(char *dst, const char *src);
char  *strncat(char *dst, const char *src, size_t n);
size_t strspn(const char *s, const char *accept);
size_t strcspn(const char *s, const char *reject);
char  *strdup(const char *s);

int atoi(const char *s);
long strtol(const char *s, char **endptr, int base);

void *malloc(size_t sz);
void *calloc(size_t nmemb, size_t sz);
void *realloc(void *ptr, size_t sz);
void  free(void *ptr);

void *memcpy(void *dst, const void *src, size_t n);
void *memmove(void *dst, const void *src, size_t n);
void *memset(void *base, int byte, size_t n);

void  qsort(void *base, size_t nelems, size_t width,
            int (*compar)(const void *, const void *));
void *bsearch(const void *key, const void *base, size_t nelems, size_t width,
              int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *p_nelems, size_t width,
            int(*compar)(const void *, const void *));
void *lsearch(const void *key, void *base, size_t *p_nelems, size_t width,
              int(*compar)(const void *, const void *));

char *gets(char buf[]);
char *fgets(char buf[], int buflen, FILE *fp);
int   fgetc(FILE *fp);
```