

CS240 - C Programming Laboratory

Exam #2

2004 Mar 24

“There’s 10 types of people in this world:
those who understand binary and those who don’t.”

Instructions: Read carefully through the entire exam first, and plan your time accordingly. Note the relative weights of each segment as a percentage of the total exam score.

This exam is **closed book, closed notes**. You may *not* refer to any books or other materials during the exam.

Write your answers on this exam. You may use both sides of the page.

When you are done, present your completed exam and your student identification to the instructor or proctors at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

Name:

Student Number:

Signature:

1. (Pointers - 20%)

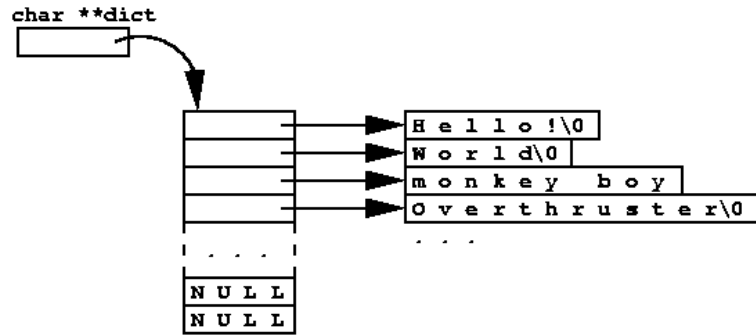
The Humanoid Resources Department of your new employer has contracted your help in constructing a software translator between English and the native tongue of the company's alien masters, Lectroidian. Toward that end, you must write a sorting routine that takes in a dictionary of English strings and returns that dictionary in sorted order.

Write a function `stringNCmp()`, which given two string pointers and a maximum length `n`, compares the first `n` characters of the strings. The comparison should be case-sensitive, and should use the ordering of the system's character set to determine which characters are *less* than others. On the next page, you will use this function to sort the dictionary.

```
/** stringNCmp
 * The stringNCmp() function compares the first (at most) n characters
 * of s and t. It returns an integer less than, equal to, or greater
 * than zero if s is found, respectively, to be less than, to match,
 * or be greater than t.
 * Examples:
 *     stringNCmp("alpha", "baker", 30) < 0
 *     stringNCmp("baker", "baker", 30) == 0
 *     stringNCmp("charlie", "baker", 30) > 0
 *     stringNCmp("charlie", "charles", 5) == 0
 */
int stringNCmp(const char *s, const char *t, size_t n)
{

    int i = 0;
    for (i = 0; (s[i] == t[i]) && (i < n); i++)
        if (s[i] == '\0') return 0;

    if (!(i < n)) return 0;
    return (s[i] - t[i]);
}
```



The input to your `sortDict()` function is a pointer to an array of pointers to character strings, (`char **`). The strings have already been read into the array in arbitrary order. There are guaranteed to be at most `MAXSTRS` strings in the array, and any pointers leftover in the array are initialized to `NULL`. It is not guaranteed that all of the strings are properly null-terminated, but your specification is to assume that no legal string can be longer than `MAXSTRLEN` characters.

Using the `stringNCmp()` function specified on the previous page, (you may assume that you have a correct implementation,) write the `sortDict()` function below. You may use any sorting algorithm you like. Do not worry about sorting duplicate strings.

```

/** sortDict()
 * Sort the dictionary into lexicographic (dictionary) order.
 */
void sortDict(char **dict)
{

    // Simple selection sort.
    int sorted = 0, least = 0, i = 0;
    char *temp = NULL;

    for (sorted = 0; (sorted < MAXSTRS) && (dict[sorted])); sorted++)
    {
        least = sorted;
        for (i = sorted + 1; (i < MAXSTRS) && (dict[i] != NULL); i++)
        {
            if (stringNCmp(dict[i], dict[least], MAXSTRLEN) < 0)
                least = i;
        }
        temp = dict[sorted];
        dict[sorted] = dict[least];
        dict[least] = temp;
    }
}

```

2. (Testing - 15%)

As part of your new job with Yoyodyne Propulsion Systems, Inc., you are given a prototype executable component, and asked to perform “black-box” unit testing on the program.

The specifications you are given for the program are simply this: read in a stream of input from `stdin`. The input is expected to be groups of integers such that the first integer is the count of subsequent integers on the rest of the line. The output (on `stdout`,) should be the mean and median of each line of integers. Upon reaching the end of the input stream, the program should print out the grand total of all of the integers. The specification states that on any errors, (including input and overflow errors,) the program should halt and print an appropriate error message.

List the major kinds of testcases you would use to check this program for bugs. For example, the test suite should include positive testcases with an odd number of identical integers, “ x ”; the mean and median value for those cases should be “ x ”.

Positive Tests:

- Input with all integers the same; mean and median should be same.
- Input with even and odd numbers of integers – check median value.
- Input with all negative integers.
- Input with all positive integers.
- Input with all zeros.
- Input with all negative ones.

Negative Tests:

- Empty line input.
- Negative number as line count.
- Input with non-numeric characters in the integers.
- Input with integers larger than `MAXINT`.
- Input where grand total is greater than `MAXINT`

Boundary Cases:

- Negative number as line count.
- Input line of “0”.
- Input line of “1”, followed by single integer.
- Input line of `MAXINT` integers.

3. (Pseudo-Dynamic Allocation - 20%)

The spacecraft being constructed by your new employers is intended to breach the Eighth Dimension, and therefore will not be running Unix. Instead of relying on the `malloc()` and `free()` system calls, the onboard systems will require a pseudo-dynamic memory allocator, which you have been asked to provide.

As demonstrated in lecture, some architectures fail with a “bus error” if pointers in the system are not properly word-aligned. Write an allocator function below, which does out properly word-aligned segments of a statically-allocated array of bytes. Your code should be as portable as possible – you don’t know what the word size on Yoyodyne’s specialized hardware will be, but you may assume that a memory word is the storage size of an integer in the system.

```
// Here is the byte array we will pseudo-dynamically allocate from.
static char allocbuf[ALLOCSIZE];
// Use this pointer to point to the next available byte of memory.
static char *allocp = allocbuf;

/** alloc()
 * Returns a character pointer to a chunk of at least 'n' contiguous,
 * word-aligned bytes from the statically-allocated buffer above.
 * Returns NULL if request cannot be satisfied.
 */
char *alloc(int n)
{
    if (0 != (n % sizeof(int)) )
        { n += sizeof(int) - (n % sizeof(int)); }
    if ((allocbuf + ALLOCSIZE - allocp) >= n)
    {
        allocp += n;
        return allocp - n;
    }
    else
        return NULL;
}
```

Also write the matching deallocator function for your system. Your employers have made the ridiculous assertion that they will be careful to always deallocate memory in exactly the opposite order in which they requested memory from your `alloc()` function. You may therefore assume that it is safe to deallocate all of the memory above the provided pointer. It is not practical to check whether or not the pointer passed to your deallocator function actually came from your allocator, but you can check whether or not the pointer belongs in the range of your statically-allocated array of bytes, and whether it is word-aligned.

Your function should do nothing if passed a pointer not within the array. If passed an unaligned pointer, deallocate only as much memory as is safe for your system to continue working properly.

```
/** afree()
 * Deallocates pointers allocated by alloc(). Assumes that all pointers
 * above us in the array can also be deallocated. Does its best if
 * given a pointer that wasn't from alloc(), or an unaligned pointer.
 */
void afree(char *p)
{
    if (0 != (((int)p) % sizeof(int)) )
        { ((int)p) += sizeof(int) - (((int)p) % sizeof(int)); }
    if ((p >= allocbuf) && (p < allocbuf + ALLOCSIZE))
        allocp = p;
}
```

4. (Command-line args - 15%)

Dr. Lizardo, (your new boss,) demands a program that takes three possible command-line parameters: “-double”, “-half”, and “-negate”. The program then reads in a stream of positive integers from standard input, (using a `getInt()` function that is already provided for you,) performs the requested transformations on the integers, and writes them to standard output. The command-line options may occur multiple times, in any combination, and in any order.

For example, executing with the parameters, “-double -negate -double” with an input stream of “1 2 34”, would produce “-4 -8 -136” as output.

You may completely ignore unrecognized command-line parameters, and ignore potential overflow errors. Read integers until `getInt()` returns less than or equal to 0.

```
/** getInt() -- Assume this is already implemented for you.
 * Returns non-negative integers read from standard in.
 * Returns -1 on errors or if no more input.
 */
int getInt();

/** stringNCmp() -- See Q1. Assume it works properly. */
int stringNCmp(const char *s, const char *t, size_t n);

int main(int argc, char **argv)
{
    int x = 0;
    int param = 0;

    while ((x = getInt()) > 0)
    {
        param = 1;
        while (param < argc)
        {
            if (stringNCmp("-double", argv[param], 8) == 0)
                { x *= 2; }
            else if (stringNCmp("-half", argv[param], 6) == 0)
                { x /= 2; }
            else if (stringNCmp("-negate", argv[param], 8) == 0)
                { x *= -1; }
            param++;
        }
        printf("%d ", x);
    }
    printf("\n");
}
```

5. (Rectangle Sorter - 30%)

In order to complete their Oscillation Overthruster, the research and development department at Yoyodyne requires a high-speed rectangle-sorter.

Write a program that stores rectangles in a sorted binary tree. You must `typedef` a `Rect` structure to hold the width and height of each rectangle, and a `RectNode` structure that allows instances of the `Rect` structure to be stored in binary trees. Rectangles are sorted according to area.

You must implement the four functions prototyped here:

```
Rect *ralloc(int width, int height);
    // Allocates a fresh rectangle with specified dimensions.
RectNode *add(Rect *r, RectNode *head);
    // Add new rectagle r into binary search tree rooted at head.
void print(RectNode *head);
    // Print out all rectangles in tree in order.
void destroy(RectNode *head);
    // Reclaim all space used by tree for Rect's and RectNode's.
```

Given a main program for testing like the one below,

```
int main()
{
    RectNode *root = NULL;

    root = add(ralloc(5,6), root);
    root = add(ralloc(2,7), root);
    root = add(ralloc(4,8), root);
    root = add(ralloc(1,2), root);
    root = add(ralloc(3,4), root);
    root = add(ralloc(5,6), root);

    print(root);

    destroy(root);
}
```

we would expect the output,

```
(1,2)
(3,4)
(2,7)
(5,6)
(5,6)
(4,8)
```

You may find it helpful to define other helper functions.

```

// Define your structs

typedef struct
{
    int width;
    int height;
} Rect;

typedef struct rectnode
{
    Rect *r;
    struct rectnode *left;
    struct rectnode *right;
} RectNode;

int rectcmp(Rect *r1, Rect *r2)
{ return (r1->width * r1->height) - (r2->width * r2->height); }

// Allocates a fresh rectangle with specified dimensions.
Rect *ralloc(int width, int height)
{

    Rect *temp = (Rect *)malloc(sizeof(Rect));
    temp->width = width;
    temp->height = height;
    return temp;
}

// Print out all rectangles in tree in order.
void print(RectNode *head)
{

    if (head)
    {
        print(head->left);
        printf("(%d,%d)\n", head->r->width, head->r->height);
        print(head->right);
    }
}

```

```

// Add new rectagle r into binary search tree rooted at head.
RectNode *add(Rect *r, RectNode *head)
{

    if (! head)
    {
        head = (RectNode *)malloc(sizeof(RectNode));
        head->r = r;
        head->right = NULL;
        head->left = NULL;
        return head;
    }
    if (rectcmp(r, head->r) < 0)
        head->left = add(r, head->left);
    else
        head->right = add(r, head->right);
    return head;
}

```

```

// Reclaim all space used by tree for Rect's and RectNode's.
void destroy(RectNode *head)
{

    if (head)
    {
        destroy(head->left);
        destroy(head->right);
        if (head->r)
        {
            free(head->r);
            head->r = NULL;
        }
        free(head);
    }
}

```