

CMSC 212 Midterm #2 (Spring 2005)

Solutions

Name _____

Signature _____

Discussion Section Time (circle one):

12:00 1:00 2:00 3:00
Asad Konstantin

- (1) This exam is closed book, closed notes, and closed neighbor. No calculators are permitted. Violation of any of these rules will be considered academic dishonesty.
- (2) You have 75 minutes to complete this exam. If you finish early, you may turn in your exam at the front of the room and leave. However if you finish during the last ten minutes of the exam please remain seated until the end of the exam so you don't disturb others. Failure to follow this direction will result in points being deducted from your exam.
- (3) Write all answers on the exam. If you need additional paper, we will provide it. Make sure your name is on any additional sheets.
- (4) Partial credit will be given for most questions assuming we can figure out what you were doing.
- (5) Please write neatly. Print your answers, if that will make your handwriting easier to read. If you write something, and wish to cross it out, simply put an X through it. Please clearly indicate if your answer continues onto another page.

Question	Possible	Score
1 a & b	8	
1 c & d	8	
2	21	
3	18	
4a	10	
4b	15	
5	20	
Total	100	

1.) [18 points] Define and explain the following terms:

a) Describe the purpose of `#ifndef` explain how it can be used to manage the inclusion of header files

- `ifndef` is a preprocessor directive
- it will consider the lines between the `ifndef` statement and the corresponding `endif` if and only if the variable named as the `ifndef`'s argument is not defined
- it can be used to manage the inclusion of header files by having the `#include` inside of an `ifndef/endif` pair - along with the corresponding `#define` of the variable named; this will allow the `#include` to be done once and only once
- any code that is inside of the `ifndef/endif` pair will only be processed if the variable given as an argument to the `ifndef` is not defined when the line is processed

b) Explain the similarities and differences between the two function prototypes given here:

```
int func1(int* const a);
```

```
int func2(const int* a);
```

- they both pass one integer pointer as the one and only parameter.
- the `func1` says that the pointer itself is a constant and `func2` says that the integer pointed at is a constant.
- In the function corresponding to the first one, `a = malloc(sizeof(int))` would not be allowed and in the second one, `*a = 7` would not be allowed.

c) Compare and contrast (both differences and similarities) between `malloc`, `calloc` and `alloca`.

- All three are ways to request memory to be allocated for the program, all three require the amount of memory being requested as an argument, and all three return a pointer to that space as the return value of the function.
- `alloca` is different from the other two in that the memory allocated comes from the stack rather than the heap
- `calloc` is different from `malloc` in that `calloc` requires 2 arguments (both the size of a single item and a count of the number of those items being requested) - it is also different in that it initializes all of the memory spaces to 0

d) Briefly explain how floating point values are stored in binary.

- There must be a mantissa and an exponent.
- The mantissa and exponent are both integer form - where the mantissa is normalized so you know that the decimal point is assumed to be a certain place (even though it does not appear) and the exponent tells how far one way or the other to move that decimal point.
- There must be a method by which to indicate each is positive or negative (individually) - the mantissa usually has a sign-bit and the exponent usually uses an offset.

- 2.) [21 points] Write the complete program (using any standard libraries you like) that will determine if any of the command line arguments are also the keyword (name) identifying an environment variable in the current shell. If it does match a variable's keyword, it should print as described below. If it does not match, an environment variable should be added to the list of environment variables with the keyword specified on the command line as its name and "prog" as its value and print as defined below. You can assume it matches if and only if the environment variable is exactly the same case as what was typed, but you also must assume you don't know the number of arguments or the length of any argument typed. If the same argument is passed more than one time on the same command line it should process once for each time it appears. The user may have accidentally typed a dash immediately before the argument; if the dash was typed it should be ignored in the comparison as if the dash had not been typed.

Output: If it was added, you must display exactly what was added (i.e. if Jan was typed as an argument and Jan is not an environment variable "Jan=prog added" should be printed). If the argument was found in the list of environment variables, you should print it as it was found in the list (i.e. if Jan=me is in the list of environment variables and Jan is given as an argument, Jan=me should be printed).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

int main(int argc, char* argv[], char *envp[]){
    int i;
    char *val;
    char *cpy;
    int tmp;

    for (i = 1; i < argc; i++){
        cpy = malloc(strlen(argv[i] + 1));
        if (cpy == NULL) exit(1);
        if (*argv[i] == '-')
            strcpy(cpy, argv[i]+1);
        else
            strcpy(cpy, argv[i]);
        if ((val = getenv(cpy)) != NULL)
            printf("%s=%s\n", cpy, val);
        else{
            val = malloc(strlen(cpy)+6);
            if (val == NULL) exit(1);
            strcpy(val, cpy);
            strcat(val, "=prog");
            tmp = putenv(val);
            if (tmp == 0)
                printf("%s added\n", val);
            else
                printf("error in adding %s\n", val);
            free(val);
        }
        free(cpy);
    }
    return 0;
}
```

}

- 3.) [18 points] Write the output from the following program. Assume the program does run all of the way through without a segmentation fault. If there are places that you do not know the actual value being printed, you should write that value in the output as "???". This uncertainty could occur because of uninitialized values being used or because of pointers into unallocated space.

```
#include <stdio.h>
#include <malloc.h>

static int stint;

void funct(int a){
    /* you don't know what happens in this function*/
    /* but no output takes place */
}

int main(void){
    int a = 5, *b = &a, c[] = {10,21,12};
    int *d, *e, f;

    printf("%d %d %d \n",*b, *c+1, stint);
    e = b;
    b = malloc(sizeof(int));
    d = malloc(sizeof(int));
    *d = 7;
    *b = 9;
    *e = 11;
    printf("%d %d %d\n",*e, *d, a);

    funct(a);
    e = b;
    free(b);
    printf("%d %d %d\n",*e, f, a);
    return 0;
}
```

_____OUTPUT_____ alignment shown here just to make it easier to read _____

```

5      11      0
11     7       11
???   ???     11
```

4.) [25 points] In project Project #3, Recall ast.h continued:

```
typedef enum { operatorNode, variableNode, constantNode } nodeType;
typedef enum { plusOperator, minusOperator, multOperator, divOperator } operatorType;
```

```
typedef struct _node {
    nodeType type;
    struct _node *left;
    struct _node *right;
    int value;
    operatorType operator;
    char *name;
} ASTnode;
```

```
ASTnode *createOperatorNode(operatorType op, ASTnode *left, ASTnode *right);
ASTnode *createConstantNode(int constant);
ASTnode *createVariableNode(char *name);
ASTnode *copyTree(ASTnode *node);
```

- a) You suspect a bug in the copyTree code is due to left and right sub-tree getting switched during the copy, write a test to verify if this is the case. Your test should return 0 if copyTree is working correct, and -1 if there is a problem. You may use the functions listed above (and you may assume you have already tested to know that these create functions listed above work correctly).

```
int testcopytree(void) {
    ASTnode *origtree = NULL;
    ASTnode *newtree = NULL;
    ASTnode *lchild = NULL;
    ASTnode *rchild = NULL;

    lchild = createConstantNode(5);
    rchild = createConstantNode(7);
    origtree = createOperatorNode(minusOperator, lchild, rchild);

    newtree = copyTree(origtree);
    if ((origtree -> right -> value == newtree -> right -> value)
        &&(origtree -> left -> value == newtree -> left -> value))
        return 0;
    else
        return -1;
}
```

b) Write a function that given two AST trees returns 0 if the trees are structurally equal and -1 if they are not. Structurally equal for a given node in a tree means that the nodes are of the same type and in the case of constant and variable nodes have the same value or strings that are equal. For operator nodes, they must have the same operator and structurally equivalent sub-trees.

```
int isequal(ASTnode *t1, ASTnode *t2){

    /*both NULL so they are structurally equal*/
    if(t1 == NULL && t2 == NULL) return 0;

    /* one is NULL other is not so aren't structurally equal */
    if (t1 == NULL && t2 != NULL) return -1;
    if (t1 != NULL && t2 == NULL) return -1;

    /* both are non-NULL - need to look at the contents of the node*/
    /*roots of diff types so can't be structurally equal */
    if (t1 -> type != t2 -> type) return -1;

    /* both of the same type so they must be one of the three*/
    if (t1->type == constantNode &&
        t1 -> value != t2->value) return -1;
    else if (t1->type == variableNode){
        if (t1.name == NULL && t2.name == NULL) return 0;
        if (t1.name == NULL && t2.name != NULL) return -1;
        if (t1.name != NULL && t2.name == NULL) return -1;
        strcmp(t1->name,t2->name) != 0) return -1;
    }else if (t1 -> type == operator Node){
        if (t1->operator != t2->operator) return -1;
        if (isequal(t1->left, t2->left) != 0) return -1;
        if (isequal(t1->right, t2->right)!=0) return -1;
    }
    return 0;
}
```

5.) [20 Points] Consider a function `apply` that takes three parameters, the first is a pointer to a function called `binaryFunc`, the second is an array of integers, and the third is an integer that is the number of elements in the array of integers. The `binaryFunc` function takes two integer parameters and returns an integer result. The function `apply` should call the passed `binaryFunc` function on the first two elements of the passed array, for each additional element of the array it should call the `binaryFunc` on that array element and the previous return value of the `binaryFunc` function. If there are less than two elements in the passed array, `apply` should return `-1`, otherwise it should return the return value of the final call to `binaryFunc`.

a) Write a typedef for a type `binaryFuncType` which is a pointer to a function described above for `binaryFunc`.

```
typedef int (*bfunctType)(int, int);
```

b) Write the function `apply`. Make sure to include the definition of parameters and return values.

```
int apply (bfunctType bf, int* arr, int size){
    int result = 0, cnt = 0;
    if (size < 2) return -1;
    else{
        result = bf(arr[0],arr[1]);
        for (cnt = 2; cnt < size; cnt++){
            result = bf(arr[cnt],result);
        }
        return result;
    }
}
```