

CMSC 212 Midterm #2 (Spring 2007)

Name KEY

Signature _____

University ID

GLUE/LDAP

UID: _____

Login ID: _____

Discussion Section Time (circle one):

10:00 11:00 12:00 1:00

Jose

Cole

- (1) This exam is closed book, closed notes, with no other references. No calculators are permitted. Violation of any of these rules will be considered academic dishonesty.
- (2) You have 80 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 paper itself. If you need additional paper, we will provide it. Make sure your name is on any additional sheets and that you submit all sheets you are given.
- (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.
- (6) The CMSC 212 Final Exam is scheduled: 5/12/07, Saturday: 4 - 6pm Location (as of right now) PHY 1412

Page	Question	Possible	Score
2	1 (a & b)	8	
3	1 (c & d)	8	
4	2 a	13	
5	2 b	15	
6	2 c	15	
7	3	15	
8	4	16	
9	5	10	
	Total	100	

1.) [16 points] Define and explain the following terms:

- a) Compare and contrast (both differences and similarities) "black box testing" and "white box testing".

Compare: both testing methods have the purpose of finding errors in the code you have written and/or inconsistencies between the written code and the specifications.

Contrast: In white box testing you can use the fact that you know the implementation while you have no knowledge of the implementation in black box testing.
-or- In white box testing you attempt to have maximum code coverage by testing all lines in the implementation while in black box testing you are determining if all output possibilities match what would be expected.

- b) Compare and contrast (both differences and similarities) a macro that is defined using the #define and a function (assume that the two do the same general task).

Compare: both will allow you to take arguments and do operations on those arguments returning a value

Contrast: The macro gets replaced into the code by the preprocessor while the function is connected to the call at link time -or- The macro is a character replacement method and so does no type checking/conversion while the function method can be tested for consistencies by the function header/prototype -or- Since macros are text replaced, parameters to macros can be types, structs, etc which is not possible with function params which must be variables.

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

Compare: both will allocate memory from the heap -or- both take the amount of space needed as the argument and return a `void*` pointer to that space

Contrast: `malloc` does not initialize the space allocated while `calloc` does -or- `calloc` takes two arguments (multiplying them to know the amount of space) while `malloc` only takes one

- d) Give at least two practical benefits to using dynamically linked libraries rather than statically linked libraries.

1: Only one copy of the library needs to be on the system not one for each executable that uses it

2: The library is linked at run time and so can be changed up until that time (more flexibility)

3: The library is linked at run time and so the underlying implementation of the library's functions can be changed up until that time

- 2.) [43 points] Assume the typedef's below and it is used to store a binary search tree. The three parts of this question all deal with the same definitions (repeated at the top of each page). You may define and use any helper functions you need. Read through all three parts of this question before writing any of them. (typedef declarations are written in columns just to give more room on the page for your answer)

<pre>typedef void (*pfuncy)(void *); // prints to standard output typedef int (*cfuncy)(void*,void*); // compares where 0 means equality, <0 means the first one is less and >0 means the second one is less</pre>	<pre>typedef struct _node{ void * dataitem; struct _node * left; struct _node * right; } nodety;</pre>	<pre>typedef struct _tree{ pfuncy printfunc; cfuncy compfunc; nodety * root; } treety;</pre>
--	--	--

- a) Write the complete function that will print the entire contents of the tree in ascending order.

```
void phelper(nodety * curr, pfuncy pf){
    if (curr == NULL)
        return;
    else{
        phelper(curr->left, pf);
        pf(curr->dataitem);
        phelper(curr->right, pf);
    }
}

void printtree(treety t){
    phelper(t.root, t.printfunc);
    return;
}
```

This is retyped here for the ease of your reference – it is the same as what appears on the previous page:

<pre>typedef void (*pfunc)(void *); // prints to standard output typedef int (*cfunc)(void*,void*); // compares where 0 means equality, <0 means the first one is less and >0 means the second one is less</pre>	<pre>typedef struct _node{ void * dataitem; struct _node * left; struct _node * right; } nodety;</pre>	<pre>typedef struct _tree{ pfunc printfunc; cfunc compfunc; nodety * root; } treaty;</pre>
--	--	--

- b) Write the complete function that will return 1 if the value passed as the second argument already appears in the tree and a 0 if it does not. You may assume the second argument is either NULL or is pointing at data of the same type as currently stored in the tree, and you can assume that the root of the tree is either NULL or currently pointing at the root node of a tree storing that same kind of data. Your routine should visit as few nodes in the tree as possible to compute its return value.

```
int lhhelper(nodety * curr, void *data, cfunc comp){
    if (curr == NULL) return 0;
    int cval = comp(curr->dataitem, data);
    if (cval == 0)
        return 1;
    else if (cval < 0)
        return lhhelper(curr->left, data, comp);
    else
        return lhhelper(curr->right, data, comp);
}

int datafound(treaty t, void *data){
    if (t.root == NULL || t.data == NULL) return 0;
    return lhhelper(t->root, data, t.compfunc);
}
```

This is retyped here for the ease of your reference – it is the same as what appears on the previous page:

<pre>typedef void (*pfuncy)(void *); // prints to standard output typedef int (*cfuncy)(void*,void*); // compares where 0 means equality, <0 means the first one is less and >0 means the second one is less</pre>	<pre>typedef struct _node{ void * dataitem; struct _node * left; struct _node * right; } nodety;</pre>	<pre>typedef struct _tree{ pfuncy printfunc; cfuncy compfunc; nodety * root; } treaty;</pre>
--	--	--

c) Assume you have already written a copytree function for the binary search tree defined above [note: you **do not need to implement the copytree function**, but you may assume somebody wrote it to make a copy of a tree]. You need to write a testdriver that takes in one tree as the one and only parameter, it calls the copytree function and then tests the copy. For each and every pointer, you can assume that it is either set to NULL or that it does point at an valid place in the tree. (If more than one of these error conditions are found – you only need to return the first one you find.) It returns:

- 0 if the two trees are identical (same structure and containing good copies of the same values in every position)
- 1 if the two trees are not a true copy because they share any data space (this means that either the node or the data within a node are in the same memory location)
- 2 if the two trees have exactly the same structure but a different value in one or more places in that structure (for example the root node contains a 5 in the original tree but a 7 in the copy tree)
- 3 if the two trees passed don't even have the same structure (for example the root node in the original tree has a left child in one tree but the root of the copy has no left child at all)

===== LESS BASED ON HIERARCHY IN ORDER =====

```
int thelper(node *ot, node *nt, cfuncy cf){
    int rval, lval, cval;
    if (ot == NULL && nt == NULL) cval=0;
    else if (ot == NULL && nt != NULL) cval=3;
    else if if (ot != NULL && nt == NULL) cval=3;
    else if (cf(ot.dataitem,nt.dataitem)!=0)
        cval=2;
    else if (ot == nt || ot.dataitem == nt.dataitem)
        cval=1;
    else cval = 0;
    int lval = thelper(ot->left,nt->left,cf);
    int rval = rhelper(ot->right,nt->right,cf);
    if (lval>=rval && lval>=cval)return lval;
    else if (rval>=lval && rval>=cval) return rval;
    else if (cval>=lval && cval>=rval) return cval;
}
int treecompare(treaty (oldt){
    treaty newt;
    copytree(oldt,&newt);
    return thelper(oldt.root,newt.root,oldt.compfunc);
}
```

```

===== OR Complete Hierarchy =====
int notsamestruct(node *ot, node *nt){
    if (ot==NULL && nt==NULL) return 0;
    if (ot==NULL && nt!=NULL) return 1;
    if (ot!=NULL && nt==NULL) return 1;
    if (notsamestruct(ot->left, nt->left))
        return 1;
    return notsamesruct(ot->right,nt->right);
}

int diffvalues(node *ot, node *nt, cfuncty cf){
    if (cf(ot->dataitem, nt->dataitem) != 0)
        return 1;
    else{
        if (diffvalues(ot->left,nt->right,cf))
            return 1;
        return difvalues(ot->right,nt->right,cf);
    }
}

int sharespace(node *ot, node *nt, cfuncty cf){
    if (ot == nt) return 1;
    if (ot->dataitem == nt->dataitem) return 1;
    if (sharespace(ot->left, nt->left))
        return 1;
    return sharespace(ot->right,nt->right);
}

int treecompare(treaty oldt){
    treaty newt;
    copytree(oldt,&newt);
    if (notsamestruct(oldt.root, newt.root)
        return 3;
    if (diffvalues(oldt.root, newt.root,oldt.compfunc)
        return 2;
    if (sharespace(oldt.root, newt.root)
        return 1;
    return 0;
}

```

3.) [15 points] Short answer questions

a) Assuming you want to run your program using the gdb debugger (your source file is named **this.c** and the executable file is named **that**).

1. What has to be different in the makefile to make this possible (just difference- running it with the debugger and running without) `___-g___`
2. What has to be typed at the shell command prompt to run it? `___gdb that ___`
3. What can be typed once inside of gdb to start it running? `__run, start, r___`

b) When you fork a process, what do you know if the PID has the value indicated?

1. **PID > 0** `__it is the parent (who successfully had a child)___`
2. **PID == 0** `___ it is the child_____`
3. **PID < 0** `_____ there was an error_____`

c) What are the three parts used in the storage of a floating point number and what is the purpose of each?

- i) `___sign bit ___`
`___to tell if the value is positive or negative___`
- ii) `___mantissa_____`
`___to tell the digits of the actual value stored_____`
- iii) `___exponent___`
`___to tell the power of 2 the mantissa should be___`
`___multiplied by___`

d) What is the purpose of the `$$` when it is used in a makefile?

`___a wildcard_____`

`___replaced by the target at the time of use_____`

- 4.) [16 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 whenever uninitialized variables are being used and whenever there is a pointer into unallocated space.

<pre>#include <stdio.h> #include <malloc.h> typedef int *(*fpntrty)(int); int * funct1(int a){ int *b = (int *) malloc(sizeof(int)); *b= a++ * 2; return b; } int * funct2(int a){ int b = ++a * 10; return &b; } int funct3(fpntrty f, int i){ int *a = (int*)malloc(sizeof(int)); printf("a=%d,i=%d\n",*a,i); a = f(i); printf("a=%d,i=%d\n",*a,i); return *a; }</pre>	<pre>int main(void){ int *c, d = 6; c = (int *) malloc(sizeof(int)); *c = 20; printf("c = %d,d = %d\n",*c,d); d = funct3(funct1,d); fpntrty fp = funct2; d = 2; c = fp(d); printf("c = %d,d = %d\n",*c,d); d = funct3(fp,5); printf("c = %d,d = %d\n",*c,d); return 0; }</pre>
---	---

c = 20, d = 6

a=???, i=6

a=12, i=6

c = ???, d = 2

a=???, i=5

a=???, i=5

c = ?? d = ???

- 5.) [10 Points] Write the assembly language program that will read two integers from standard input and write the sum, difference (1st – 2nd) and product of those two integers to the screen (one per line). You may assume that both values put in as input are positive and small enough that there is no overflow in any of these calculations. For a summary of the assembly language used for projects 1 and 3 see the next page.

```
    read R2
    read  R3

    add  R2  R3  R4
    write R4

    mv R4 R3
    neg R4
    add  R2  R4  R4
    write R4

    mv R5 R3
    li R6 1
    neg R6
    li R4 0
LOOP: beq R0 R5 END
      add R2 R4 R4
      add R5 R6 R5
      beq R0 R0 LOOP
END:  write R4

    halt
```

You may continue any answer onto this page – you must write on the page that contains the first part where it is continued to, and you must write on this page where it is continued from.

Assembly Language from Projects 1 and 3

- lw <reg1> <reg2> <memory>
- li <reg1> <number>
- mv <register₁> <register₂>
- sw <register₁> <register₂> <memory>
- add <register₁> <register₂> <register₃>
- neg <register₁>
- beq <register₁> <register₂> <memory>
- bal <register₁> <register₂> <memory>
- read <register₁>
- write <register₁>
- halt

DO NOT WRITE ON THIS PAGE

**NOTHING ON THIS PAGE WILL BE
GRADED AT ALL**

If you need extra room beyond the one blank page (page 10) – raise your hand and we will bring you extra paper. All pages you receive must be turned in when the papers are collected.

DO NOT WRITE ON THIS PAGE