CMSC 212 FINAL EXAM (Spring 2005)

Name _GRADING DIRECTIONS_

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 120 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	**20**	
2	**11**	
3	**15**	
4	**12**	
5	**10**	
6	**12**	
7	**20**	
Total	**100**	

1.) [20 points]

a) Explain the purpose of C's preprocessor and give two different preprocessor directives (with explanation of their purpose.

[3 pnts] Makes a pass over the program substituting pre-processor directives with values or filles from other locations.

[2 pnts] #include - include a file into the file being compiled

[2 pnts] #define - defines a value for a preprocessor macro (a constant or short bit of code).

b) What is the purpose of each of these compiler switches?

1. **[1 pnts] -o** The next switch indicates the output file for the compilation
2. **[1 pnts] -O** Turns on the optimizer which tries to generate faster code
3. **[1 pnts] -c** Indicates that the compiler should compile to a .o file, but not invoke a linker to produce an executable file
4. **[1 pnts] -g** requests the compiler to generate additional output to enable running the debugger on the resulting executable.

c) According to the code segment in the first column - write the type of each of the given expressions in the third column. (i.e. Integer, Integer pointer, address of an integer pointer, etc) If you can not tell from the code given, you must write the word "unknown". You do not need to worry about what value (or even if there is space allocated) - just what type it would be.

int a, b; int* t, q;	T	[1 pnts] Undefined, C is case sensitive and only lowercase t is defined.
	*(&a)	[1 pnts] int
	&b	[1 pnts] Pointer to int or address of an integer
	q	[1 pnts] int
	*t	[1 pnts] int

d) Given one example of a situation where optimizing wall time (vs. process time) makes more sense and one where optimizing process time (vs. wall time) makes sense.

[2 pnts] If your program is I/O bound, you should optimize wall time which is not calculated as process time.

[2 pnts] If your program is compute bound, you should optimize for process time so that the impact of other processes running on the system (measured in wall time) will not be a factor.

2.) [11 points] Give all of the output of the following code on the lines given (it does compile and run). If you have a situation where you have a pointer into non-allocated space, use ??? to indicate that value.

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

int f1(int *x){
   *x = 2;
    x = (int*)calloc(sizeof(int),1);
    *x = 3;
    return *x;
}
int* f2( int x, int **y){
  int m= 8;
  int *t = &m;
  x += m--;
  y = &t;
  printf("%d %d %d\n",x,*t,**y);

  return t;
}

int main(void){
   int a,b,c;
   int *p, *q, *r;

   a = 5;
   p = &a;
   b = f1(p);
   printf("%d %d %d\n",a,b,*p);

   c = 5;
   q = &c;
   r = f2( c,&q);
   printf("%d %d %d\n",c,*q,*r);

   return 0;

}
```

OUTPUT goes on these lines
2 3 2
13 7 7
5 5 ?
1 point for each value
up to -2 for extra values
or incorrect end of lines etc.

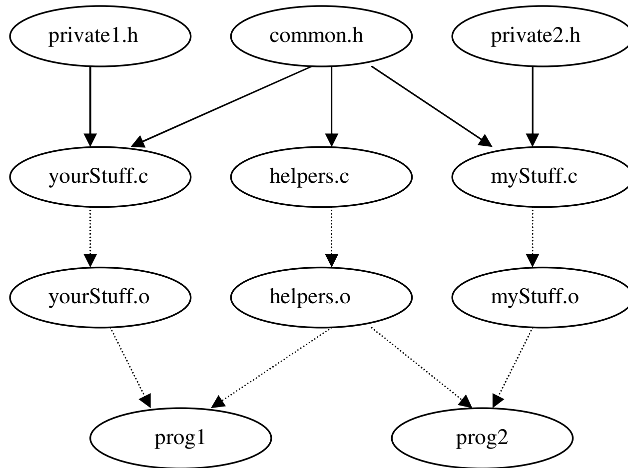
3.) [15 points] Write the assembly code for the following function (following the stack conventions of the machine):

```
void  foo(int x, y)
{
    if (x > 0) {
        foo(x-1, y);
    else if (y > 0) {
         printf("%d\n", y);
    }
}
```

```
foo:            Store R14 R15 0
                Load R10 R0 #1
                Negate R10
                Add R15 R10 R15
                Move R11 R2
                Add R2 R10 R2
                Negate R11
                Bnn R11 skip1
                Branch R14 R0 foo
                Branch R0 R0 skip2
skip1:          Move R12 R3
                Negate R12
                Bnn R12 skip2
                Output R3
skip2:          Load R10 R0 #1
                Add R15 R10 R15
                Load R14 R15 0
                Branch R0 R14 0
```

Grading:
2 points for correctly testing and branching on x > 0
2 points for correctly storing the x and y values
2 points for storing stack pointer
3 points for initiating the recursive call
2 points for restoring stack pointer
2 points for coming out of the recursion
2 points for the output when y > 0

4.) [12 points] Write a makefile which builds the programs with the following dependencies. A solid line indicates a file included by another file and a dash line indicates a file that should be built from one or more files. You must write the makefile in such a way that typing the command make at the command line will create both executable files based on the correct versions of the files in the current directory.




```
all: prog1 prog2

yourStuff.o: yourStuff.c private1.h common.h

helpers.o: helpers.c common.h

myStuff.o: myStuff.c common.h private2.h

prog1: yourStuff.o helpers.o

      gcc -o prog1 yourStuff.o helpers.o

prog2: myStuff.o helpers.o

      gcc -o prog2 myStuff.o helpers.o
```

GRADING: 12 points total
3 points for both executables being the first directive (type make to compile both)
OK if did or didn't use the implied dependencies (not putting x.c in the list)
OK if did or didn't use the implied constructions (not putting the cc -c x.c in the list)
4 points for the .o creation directives
    -1 point for each missing or extra item listed to a maximum of 4 points
5 points for the executable creation directives
    -1 for each missing or extra item (file, option, etc) to a maximum of 5 points

5.) [10 Points] Implement strstr without using any array notation or any string library functions. (searches for the second parameter within the first and returns a pointer to the beginning of that substring as the return value of the function).

char * strstr(const char * full, const char* sub);

```
{
    char *curr1, *curr2;
    if (!full || !sub) return NULL;
    for (;*full; full++) {
          for (curr1=sub, curr2 = full; *curr1 && *curr2; curr1++, curr2++) {
                if (*curr1 != *curr2) break;
          }
          if (!*curr1) return full;
    }
    return NULL;
}
```

```
GRADING:
2 points for the NULL or NULL test (what to do if either or both are NULL)
4 points for the outer loop that traverses starting the comparison at each character
of the full string
4 points for the innerloop that compares each element of the sub string to the
corresponding elements in the full string

-4 if used array notation but otherwise it is completely working
```

6.) [12 points] Write a paragraph for each sub-question

a) Even though lcov reports 100% coverage of your program being tested, how could the program being tested still have bugs.

```
[3 points] Lcov reports statement level coverage information, but it is
possible that different paths of execution have not been exercised even though
all statements have run.
```

b) Give one benefit of **each** in C and Fortran90's choices in representing and storing arrays.

```
[3 points] C's representation is smaller and has less overhead.  Fortran90's
allows for multi-dimensional arrays and allows for runtime detection of array
bounds errors.
```

c) Give two examples of situations where you might want a Java program to call one or more functions written in C.

```
[3 points] You have a highly tuned library (like an mpeg encoder) written in C
and you want to re-use this code as part of a new Java program. You need to
call some machine specific code (i.e. to a low level graphics processor) which
can't be written in Java.
```

d) If you were to write code that was a Java function that was calling a C function, you would need to consider the possibility of them sharing data. When converting between the scalar types of Java (int for example) to the corresponding type in C - what is the primary difficulty.

[3 points] In `Java`, the int types have a well defined size (16, 32, or 64 bits), but in C the size of the int type is machine specific, so there would be concerns about the what could or could not be represented depending on the relative size of the ints.

7.) (20 points) Write the following routines:

void *myMalloc(int size);
void *myCalloc(int num, int size);
void myFree(void *ptr);

These routines must mimic the C library routines malloc, calloc and free respectively. Your are given a function void *add10Meg() that grows the memory of your process by 10 megabytes at a time, and returns a pointer to the newly added space. The only C function you may call (that you don't write) is add10Meg().

```
typedef struct  _boundry {
      int size;
      int free;
      struct _boundry *next;
} boundry;

boundry *list;
```

==

void *myMalloc(int size)

```
{
      boundry *curr, *split;

      size =  (size-1)/sizeof(boundry) + 2;

      for (curr=list; curr; curr= curr->next) {
            if (curr->free && curr->size >= size) {
                  if (curr->size > size+1) {
                        split = curr + size;
                        split->next = curr->next;
                        split->size = curr->size - size - 1;
                        split->free = 1;
                        curr->size = size;
                        curr->free = 0;
                        curr->next = split;
                  } else {
                        curr->free = 0;
                  }
                  return (curr + 1);
            }
      }

      curr = (boundry *) add10Meg();
      split = curr + size;
      split->next = list;
      split->size = (10 * 1024 * 1024)/sizeof(boundry)  - size - 2;
      split->free = 1;
      curr->free = 0;
      curr->size = size;
      curr->next = split;
}
```

Comment [JDP1]: Page: 9
This line was size > curr->size but then the next line tests is curr->size > size+1
this could never be true when the size > curr-> size

```
GRADING [10 points]:
3 points for correctly calling add10Meg when needed
3 points for searching for free space in list of free space
3 points for correctly splitting free space
1 points for returning pointer to correct location (not boundary tag)
```

void *myCalloc(int num, int size)

```
{
      int i;
      char *ptr;
      ptr = myMalloc(num * size);
      for (i=0; i < num * size; i++) ptr[i] = '\0';
      return ptr;
}
```

GRADING [5 points]:

```
1 point for returning pointer
2 points for calling myMalloc on correct size
2 points for setting space to 0
```

----

void myFree(void *ptr);

```
{
      boundry *head;
      head = ptr;
      head--;
      head->free = 1;
}
```

GRADING [5 points]:

```
3 points for finding boundary tag
2 points for setting it free
up to 2 points for errors if they tried to coalesce blocks, but
had errors.
```