**Name: ____________________** **Date: ____________________**

# CS270 Homework 5 (HW5)

Due Thursday, Nov 21 (start of class)
Homework and programming assignments are to be done individually.

## Goals

To understand C pointers and structs, and functions and the activation record:

- Basic pointer manipulation
- C pointers and functions
- C pointers and arrays
- C pointers and strings
- C pointers and structs
- Static memory allocation
- Dynamic memory allocation
- C pointers and swapping
- C pointers and efficiency
- C pointers to pointers
- Activation records
- Managing activation records, stack pointer, frame pointer, and function calls

## Instructions

You may need to write C programs for this assignment, but these do not need to be handed in. A hard copy of your written solution is to be turned in before class on the due date. Try to figure out the output of the programs before running any code, ***and only then*** compare your answer after running the program. This will maximize your learning, and ensure greater chance of success if similar questions appear on the final exam. Each question is worth 10 points.

## Late Policy

All homework assignments should be handed in at the beginning of class on the due date. Late assignments will be accepted up to 24 hours past the due date, with a penalty of 10% per 24-hour period. Late submissions should be made via email to cs270@cs.colostate.edu (.txt or .pdf files only), or by delivering the paper copy to Sanjay's office or under the door of room 340 CSB.

## Extra Credit

This assignment, which is nominally worth 3% of your final grade, will carry double the weight, thus allowing you to earn extra credit, for no extra work.

**Question 1 (10 points): Basic C Pointers**

```
int i;
float x;
int *pInteger = &i;
float *pFloat = &x;

i = 5678;
*pInteger = 1234;
x = 0.5678f;
*pFloat = 0.1243f;

printf("i = %d, %d, %d\n", i, *(&i), *pInteger);
printf("x = %f, %f, %f\n", x, *(&x), *pFloat);
```

a) What is the output of the code shown above?

b) Is there any difference between the address of a variable, and the value of a pointer to that variable?

c) What should the difference in the values of `pInteger` and `pFloat` to be? __________ bytes

**Question 2 (10 points): C Pointers and Functions**

```
void function(int i, int *j, float x, float *y)
{
        i = 5544;
        *j *= 100;
        x = 0.1234f;
        *y /= 10.0;

        printf("%d, %d, %f, %f\n", i, *j, x, *y);
}

int i = 2233;
int j = 1122;
float x = 5.678f;
float y = 2.468f;

printf("%d, %d, %f, %f\n", i, j, x, y);
function(i, &j, x, &y);
printf("%d, %d, %f, %f\n", i, j, x, y);
```

a) What is the output of the code shown above?

b) Which parameters can be changed by the function? Which cannot?

c) The function appears to modify the parameters `i` and `x`, but these values never make it out of the function. Why not?

**Question 3 (10 points): C Pointers and Arrays**

```
int iArray[4] = {11, 22, 33, 44};
int *pInteger = &iArray[0];
printf("%d %d %d %d\n", iArray[0], iArray[1], iArray[2], iArray[3]);

iArray[0] *= 2;
*(pInteger+1) *= 3;
pInteger[2] *= 4;
*(iArray+3) *= 5;

printf("%d %d %d %d\n", iArray[0], iArray[1], iArray[2], iArray[3]);
```

a) What is the output of the code shown above?

b) Are the following identical: `pInteger[1]`, `*(pInteger+1)`, `iArray[1]` and `*(iArray+1)`? Why?

**Question 4 (10 points): C Pointers and Strings**

```
char *str = "hello";
char str1[6] = {'t','h','e','r','e','\0'};

for (unsigned int i=0; i<strlen(str); ++i)
{
        printf("str[%d] = %c(%c)\n", i, str[i], *(str+i));
}

printf("str = %s\n", str);

for (unsigned int j=0; j<strlen(str1); ++j)
{
        printf("str1[%d] = %c(%c)\n", j, str1[j], *(str1+j));
}

printf("str1 = %s\n", str1);
```

a) What is the output of the code shown above?

b) Is there a string data type in C? If not, what is used instead?

c) What additional limitation does a string have that a character array does not?

**Question 5 (10 points): C Pointers and Structs**

```
typedef struct
{
	int i;
	float f;
} simple;

simple s;
simple *p=&s;

s.i = 1234;
s.f = 0.112233f;
printf("s.i = %d, s.f = %f\n", s.i, s.f);

p->i += 2345;
p->f *= 2.0f;
printf("s.i = %d, s.f = %f\n", s.i, s.f);
```

a) What is the output of the code shown above?

b) How does the **.** operator differ from the **->** operator with respect to structure access?

c) How many bytes does the *struct* defined above require on a 32-bit system?

**Question 6 (10 points): C Pointers and Static Allocation**

```
int i = 11;
int j = 12;

float x = 0.123f;
float y = 0.234f;

printf ("Values: %d, %d, %f, %f\n", i, j, x, y);
printf ("Addresses: %p, %p, %p, %p\n", &i, &j, &x, &y);
```

a) What is the output of the code shown above? You must run the code to find out.

b) Are local variables pushed onto the stack in forward order (i,j,x,y) or reverse order (y,x,j,i)?
[**Hint**: pushing data onto the stack increases the stack pointer.]

**Question 7 (10 points): C Pointers and Dynamic Allocation**

---

```
int array1[4];
int array2[4];

int *array3 = (int *)malloc(sizeof(int) * 4);
int *array4 = (int *)malloc(sizeof(int) * 4);

printf("Addresses: %p, %p, %p, %p\n", array1,array2,array3,array4);
```

---

a) What is the output of the code shown above? You must run the code to find out.

b) Why are the addresses of array1/array2 so different from array3/array4. Which memory pool is used for each allocation?

**Question 8 (10 points): C Pointers and Data Swapping**

---

```
void swap0(int x, int y)
{
	int temp = x;
	x = y;
	y = temp;
	printf("x = %d, y = %d\n", x, y);
}

void swap1(int *x, int *y)
{
	int temp = *x;
	*x = *y;
	*y = temp;
	printf("x = %d, y = %d\n", *x, *y);
}

int i = 1234;
int j = 5678;

printf("i = %d, j = %d\n", i, j);
swap0(i, j);
printf("i = %d, j = %d\n", i, j);
swap1(&i, &j);
printf("i = %d, j = %d\n", i, j);
```

---

a) What is the output of the code shown above?

b) Why does `swap0` fail to swap the values, even though it seems to have worked locally? Why does `swap1` work?

**Question 9 (10 points): C Pointers and Efficiency**

---

```
typedef struct
{
	int iArray[32];
	float fArray[32];
} large;

void f1(large l)
{
	printf("sizeof(l) = %d\n", (int)sizeof(l));
}

void f2(large *l)
{
	printf("sizeof(l) = %d\n", (int)sizeof(l));
}

large s;
for (int i=0; i<32; ++i)
{
	s.iArray[i] = i;
	s.fArray[i] = (float) i;
}
f1(s);
f2(&s);
```

---

a) What is the output of the code shown above?

b) How many bytes are required on the stack for parameter storage for `f1()`? `f2()`? Which is more efficient and why?

**Question 10 (10 points): C Pointers to Pointers**

---

```
int i = 12345;
int *p = &i;
int **pp = &p;

i = 2345;
printf("i = %d\n", i);

*p = 3467;
printf("i = %d\n", i);

**pp = 4567;
printf("i = %d\n", i);

printf("&i = %p, p = %p, pp = %p, *pp = %p\n", &i, p, pp, *pp);
```

---

a) What is the output of the code shown above? You must run the code to find out.

b) Why do **&i**, **p**, and ***pp** all point at the same address?

c) What is pointed at by the “pointer to a pointer” **pp**?

**Question 11 (40 points): Functions and the activation record**

This problem makes you interpret the assembly output of the LC-3 compiler (lcc) in order to figure out the stack convention. We supply the original C file (stack.c at http://www.cs.colostate.edu/~cs270/.Fall12/assignments/HW3/stack.c) and the assembly code (stack.asm at http://www.cs.colostate.edu/~cs270/.Fall13/Assignments/HW5/stack.asm) generated from the compiler. The C program and a fragment of the assembly file with the function code are shown below, these should be sufficient. Examine the C code and the assembly code and answer the questions, which are worth 2 points each. For the last question you must draw a picture of the stack at a certain point in the program, this problem is worth 12 points. This assignment does not require you to compile or run. Reading pages 389-392 of the textbook will help explain this assignment.

Here is the C code for the assignment, comments have been removed to save space:

```
int add(int param0, int param1) { int result;
result = param0 + param1; return (result);
}
int main(int argc, char *argv[]) {
int local0 = 1234;
int local1 = 2345;
printf("Result: %d\n", add(local0, local1)); return (0);
}
```

The assembly code generated by *lcc* for the add function in *stack.c* is shown on the next page

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;add;;;;;;;;;;;;;;;;;;;;;;;;;;;
lc3_add
;; stack entry
0:      ADD R6, R6, #-1
1:      ADD R6, R6, #-1
2:      STR R7, R6, #0
3:      ADD R6, R6, #-1
4:      STR R5, R6, #0
5:      ADD R5, R6, #-1

;; function body
6:      ADD R6, R6, #-1
7:      LDR R7, R5, #4
8:      LDR R3, R5, #5
9:      ADD R7, R7, R3
10:     STR R7, R5, #0
11:     LDR R7, R5, #0

;; stack exit
12:  STR R7, R5, #3
13:  ADD R6, R5, #1
14:  LDR R5, R6, #0
15:  ADD R6, R6, #1
16:  LDR R7, R6, #0
17:  ADD R6, R6, #1
18:  RET
```

Answer the following questions, using the variable names from the original program, or one of the following: return value, stack pointer, frame pointer, and return address. When the answer is the frame pointer, identify whether it is the frame pointer for *main()* or *add()*. Do not tell us that R7 is getting pushed or R5 getting popped, we already know that from reading the code. Be specific with names from the original C program: *local0, local1, param0, param1*, etc. Assume that the main program has pushed *param0* and *param1* before calling the *add()* function. The line numbers refer to the line numbers above in the assembly version of the lc3_add routine.

**Question 1:** The code at line 0 is making room on the stack for which value?

**Question 2:** What is getting pushed at lines 1 and 2?

**Question 3:** What is getting pushed at lines 3 and 4?

**Question 4:** What value is being setup at line 5 for which function?

**Question 5:** The code at line 6 is making room on the stack for which value?

**Question 6:** At line 7, which parameter is loaded, and from what frame pointer offset?

**Question 7:** At line 8, which parameter is loaded, and from what frame pointer offset?

**Question 8:** What is the code at line 9 doing?

**Question 9:** What is being stored at line 10, and to which frame pointer offset is written?

**Question 10:** What is being loaded at line 11, and from which frame pointer offset is it read?

---

**Question 11:** Is the instruction at line 11 redundant?

---

**Question 12:** What is being stored at line 12, and to which frame pointer offset is written?

---

**Question 13:** What is getting popped at line 14 and 16?

---

**Question 14:** What are registers R5 and R6 used for by the compiler?

R5:____________________________________________________

R6:____________________________________________________

---

**Question 15:** Draw a picture of what the stack looks like after executing line 12 of the program. Include all the values from the stack frame for add() and the parameters pushed on the stack by main(). In the left column specify the name and value of the variable or use the names Frame Pointer, Return Value, or Return Address. In the second column, if the value is a Frame Pointer, Return Value, or Return Address, tell whether it belongs to main() or add(). In the third column, show the offset from the Frame Pointer in array notation, for example FP+0, FP+1, etc. You should need exactly 6 rows for a correct answer. In the left margin, draw an arrow to the slots pointed at by the current Frame Pointer and Stack Pointer, again after executing line 12 of the program.

|  |  |  |
|---|---|---|
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |