Name: ____________________ **Date: ____________________**

CS270 Recitation 15
C Programming Exercise

CONFIDENTIAL: Answer sheet and grading criteria.

Goals

To understand all aspects of C pointers:

- 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

Instructions

You may need to write C programs for this assignment, but these do not need to be handed in. Try to guess the output of the programs before running any code, then compare your answer after running the program. This recitation is graded based on attendance and the TA will show the answers in the last ten minutes of class.

The Assignment

Question 1 (10 points): Basic C Pointers (a – 4 points, b - 3 points, c - 3 points)

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

i = 1234;
*pInteger = 5678;
x = 0.5678f;
*pFloat = 0.1234f;

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?

```
i = 5678, 5678, 5678
x = 0.123400, 0.123400, 0.123400
```

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

No, it's exactly the same.

c) What would you expect the difference in the values of pInteger and pFloat to be? **4** bytes

Consecutive addresses, separated by the size of an integer or float (order doesn't matter).

Question 2 (10 points): C Pointers and Functions (a – 4 points, b - 3 points, c - 3 points)

```
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 = 1122;
int j = 2233;
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?

```
1122, 2233, 5.678000, 2.468000
5544, 223300, 0.123400, 0.246800
1122, 223300, 5.678000, 0.246800
```

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

The j and y parameters can change since they are passed by reference (pointer), i and x cannot because they are passed by value.

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

Only the local copy is changed.

Question 3 (10 points): C Pointers and Arrays (a – 5 points, b - 5 points)

```
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?

11 22 33 44
22 66 132 220

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

Yes, because array names and pointers to the array are interchangeable.

Question 4 (10 points): C Pointers and Strings (a – 4 points, b - 3 points, c - 3 points)

```
char *str = "hello"; // automatically adds null termination
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?

```
str[0] = h(h)
str[1] = e(e)
str[2] = l(l)
str[3] = l(l)
str[4] = o(o)
str = hello
str1[0] = t(t)
str1[1] = h(h)
str1[2] = e(e)
str1[3] = r(r)
str1[4] = e(e)
str1 = there
```

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

No, just character arrays.

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

String must be null terminated, character arrays do not.
No null characters within a string, characters can have all sorts of weird control characters.
String can be treated as a character array, but not all character arrays can be used as strings.

Question 5 (10 points): C Pointers and Structs (a – 4 points, b - 3 points, c - 3 points)

```
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?

s.i = 1234, s.f = 0.112233

s.i = 3579, s.f = 0.224466

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

The . symbol is for normal structure access, -> is for structure access via a pointer.

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

8 bytes (for 32-bit or 64-bit systems)

Question 6 (10 points): C Pointers and Static Allocation (a – 5 points, b - 5 points)

```
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.

```
Values: 11, 12, 0.123000, 0.234000
Addresses: 0x7fff6b820b1c, 0x7fff6b820b18, 0x7fff6b820b14, 0x7fff6b820b10
```

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

Forward order (i,j,x,y), first parameter has the highest address, last parameter has the lowest address, not necessarily the same as pushing parameters. Order is fairly arbitrary anyway!

Question 7 (10 points): C Pointers and Dynamic Allocation (a – 5 points, b - 5 points)

```
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.

Addresses: 0x7fff6b820b00, 0x7fff6b820af0, 0x1929010, 0x1929030

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

The array1/array2 allocation is on the stack (static) , array3/array4 is from the heap (dynamic).

Question 8 (10 points): C Pointers and Data Swapping (a – 5 points, b - 5 points)

```
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?

```
i = 1234, j = 5678
x = 5678, y = 1234
i = 1234, j = 5678
x = 5678, y = 1234
i = 5678, j = 1234
```

b) Why does swap0 fail to swap the values, even though it seems to have worked locally? Why does swap 1 work?

The swap0 function has parameters passed by values, so it can only swap the copies. The swap1 function has parameters passed by reference (pointer), so it can swap the actual values.

Question 9 (10 points): C Pointers and Efficiency (a – 5 points, b - 5 points)

```
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?

```
sizeof(l) = 256
sizeof(l) = 8
```

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

Same as above, 256 bytes for f1(), 8 bytes for f2(), passing the pointer is more efficient since it requires less data to be copied to the stack.

Question 10 (10 points): C Pointers to Pointers (a – 4 points, b - 3 points, c - 3 points)

```
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.

```
i = 2345
i = 3467
i = 4567
&i = 0x7fff6b820b14, p = 0x7fff6b820b14, pp = 0x7fff6b820b08, *pp = 0x7fff6b820b14
```

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

&i is the address of i, p is a pointer to i, and *pp is the same as p, thus all are identical pointers to i.

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

The double pointer pp points at the pointer p, so *pp = p = &i.