Name: ______________________________________

- INSTRUCTIONS:
  - Show your work to receive partial credit.
  - Keep your eyes on your own paper and do your best to prevent anyone else from seeing your work.
  - Do NOT communicate with anyone other than the professor/proctor for ANY reason in ANY language in ANY manner.
  - This exam is closed notes, closed books, no calculator.
  - Turn all mobile devices off and put them away now. You cannot have them on your desk.
  - Write neatly and clearly indicate your answers. What I cannot read, I will assume to be incorrect.
  - Stop writing when told to do so at the end of the exam. I will take 5 points off your exam if I have to tell you multiple times.
  - Academic misconduct will not be tolerated. Suspected academic misconduct will be immediately referred to the Rollins Honor Council. Penalties for misconduct will be a zero on this exam, an F grade in the course, and/or other disciplinary action that may be applied by the Rollins Honor Council.
- TIME: This exam has 6 questions on 11 pages including the title page. Please check to make sure all pages are included. You will have 75 minutes to complete this exam.

On my honor, I have not given, nor received, nor witnessed any unauthorized assistance on this work. Also, I have read and understand the above policies for this exam.

Signature: ______________________________________

Question	Points	Score
1	23	
2	11	
3	9	
4	10	
5	7	
6	10	
Total:	70	

1. Base Conversions: Convert the following numbers.

(a) (2 points) 48_{10} to 8 bit binary (base 2)

Solution: 00110000

(b) (2 points) -48_{10} to 8 bit sign-magnitude binary.

Solution: `10110000`
Use positive value in part (a), and change sign bit to 1 to represent negative number.

(c) (2 points) -48_{10} to 8 bit 2's complement binary.

Solution: `11001111 + 1 = 11010000`

Use positive value in part a, flip all bits, add 1.

(d) (2 points) 100111011110_2 to octal (base 8)

Solution:

```
bin:100 111 011 110
oct: 4   7   3   6
```

Common error: grouping by 4 bits instead of 3 bits. You group by 4 to transform to hex, group by 3 for octal.

(e) (3 points) $AD5_{16}$ to binary (base 2)

Solution:

```
 A    D    5
1010 1101 0101
```

(f) (4 points) Encode "Ab!" as a C-style string. Give your answer in hex represenation.

Solution: 0x41622100

Common mistake: forgetting the null termination at the end of the string.

(g) (4 points) -17.625_{10} to IEEE single precision (32 bit) floating point decimal number.

Solution: negative so sign bit is 1

17.625 == 10001.101 in binary. Normalize, it becomes $1.0001101x2^4$. Encode 4 in excess-127 code (see last page for conversion), and your exponent is 10000011.

Then drop your leading 0 and the significand is 00011010000000000000000.

So: `1 10000011 00011010000000000000000` is the entire 32 bit pattern.

(h) (4 points) Add the two following integers which are represented in 8-bit 2's complement format. Then state whether or not an overflow occurs.

```
  0111 1001
  1100 1011
-----------
```

Solution: Answer is 0100 0100. I gave credit regardless of whether or not you showed the carry out bit.

An overflow does NOT occur. Adding a negative number and a positive number can never result in an overflow in 2's complement. The only way to get an overflow is to add two positive numbers, but end up with negative or vice versa.

You can also sanity check your answer: The top number is 121, the 2nd number is -53. Adding them results in 68 which is the value of the answer you calculated.

Common mistake: leaving out part of the answer – either not showing the results of the addition or not stating whether or not the overflow occurred.

2. Code Snippets. For each of the following prompts, write a snippet of code (no need for a complete function or program) which accomplishes the task. You can choose variable names unless otherwise specified in the prompt.

(a) (2 points) Write a statement which calculates the number of elements in an array of `int`s and assigns the value to a variable.

Solution: `int size = sizeof(array)/sizeof(int);`
or `int size = sizeof(array)/sizeof(array[0]);`

Common mistake: just doing `sizeof(array)` which gives you the number of bytes allocated to the array but not the number of elements in the array.

(b) (1 point) Rewrite the following code using a character string:

```
char word[] = {'F','a','l','l','!','\0'};
```

Solution:

`char* str = "Fall!";` or `char str[] = "Fall!";`

Common errors:
- combining both the forms of the declaration together (eg. `char* str[] = ...`)
- leaving the array curly braces in place and using quotation marks (eg. `char str[] = {"Fall!"};`)

(c) (4 points) Write the code to compute the sum of all the values in a 6x6 matrix of integers: `int matrix[6][6]`

Solution:

```
int row;
int col;
int sum = 0;
for(row = 0; row < 6; row++) {
  for(col = 0; col < 6; col++) {
    sum += matrix[row][col];
  }
}
```

(d) (4 points) Write code to prompt the user to enter their first **AND** last names. Then read and store their name into a character array.

Solution: This problem is largely taken from the cipher HW.

```
printf("Enter your name: ");
char name[30]; //any reasonable size ok
fgets(name, sizeof(name), stdin);
```

Scoring:
+1 for prompt
+1 for buffer declaration
+2 for correct reading of input

Common errors: Note, you had to use `fgets` for this problems! `scanf` only reads up the space. If the user entered “Valerie Summet”, `scanf` will only store “Valerie” while `fgets` would store everything (including spaces).

3. Explain things to me.

(a) (4 points) In C, we often discuss *buffer overflows*. Explain what a buffer overflow is and why it would be considered a security bug. You can include code snippets to illustrate your examples if you feel the need.

Solution: No bounds checking on arrays (buffer overflows). This means that the programmer needs to make sure not to go beyond the end of the array. Storing the length of the array into a variable and then using that variable in all situations relating to the array is a good technique.

It's considered a security bug because buffer overflows allow a program to read memory which it shouldn't be accessing. If something secure (for example, an decrypted password) is stored in memory, it could potentially be obtained via a buffer overflow.

Writing too many bytes is less of a security concern, but is still an issue. It will most likely make your program crash.

(b) (2 points) Explain why the sign-magnitude representation is considered inferior to the 2's complement representation of signed integers.

Solution: Lots of people said, ”Because it can represent fewer numbers”. This is true, but only 1 number less than 2's complement. A better answer is that it has 2 representations of 0 or that arithmetic/binary addition is significantly harder with sign magnitude.

Common issue: Explaining what sign magnitude was but not saying why it was inferior to 2's complement.

(c) (3 points) What do we mean when we say that a computer is *byte-addressable*?

Solution: Each memory address stores 1 byte (8 bits) of data.

Common mistake: saying something like ”Each address is made up of bytes” or ”Each address is 1 byte”.

4. Assume the following program compiles and runs to completion.

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

int main(void) {

  int my_nums[] = {20, -3, 42, 11};
  int* array_ptr = my_nums;
  (*array_ptr)++
  printf("1: %d\n", *array_ptr);
  array_ptr++;
  printf("2: %p\n", array_ptr);
  array_ptr++;
  *array_ptr = 30;
  printf("3: %d\n", my_nums[2]);

  char greet[] = "Hello";
  char* char_ptr = greet;
  (*char_ptr)++;
  printf("4: %c\n", *char_ptr);
  char_ptr++;
  printf("5: %c\n", *char_ptr);

  return EXIT_SUCCESS;
}
```

(a) (5 points) Give the output of the following program. If output is unknown, you can simply describe as much as you can about the data or why it is unknown.

Solution: This problem is mostly drawn from our in-class worksheet on pointers.

```
1: 21
2: unknown -- prints address of my_nums[1]
3: 30
4: 'I'   (could have used ASCII chart on the reference page to determine this)
5: 'e'
```

Scoring: 1pt per line of output.

(b) (5 points) Using what you know about the size of characters in C, draw a diagram of memory for the variables `greet` and `char_ptr` **after** the code completes. You can choose any address range you want, but be sure to clearly label each memory location. You can add more rows if you need to.

Variable	Memory Address	Value
greet	1000	I
	1001	e
	1002	l
	1003	l
	1004	o
	1005	\0
char_ptr	2000	1001

Solution: The addresses you chose could vary, but for full credit, you had to show that:

- `char`s are 1 byte in C
- elements of `greet` are stored sequentially
- the string is null terminated and that takes 1 byte
- `char_ptr` is not `greet[1]` but is instead stored separately
- the value of `char_ptr` is the address of `greet[1]`

5. Professor Summet is trying to write a function which will add 5 to the integer referenced by the parameter. Unfortunately, her program isn't doing what she wants:

```
//** Prof. Summet's version **//
void add5(int* x) {
  int y = 5 + *x;
  x = &y;
}

int main() {
  int var = 10;
  add5(&var);
  printf("result is %d\n", var);

  return 0;
}
```

(a) (1 point) What value for `var` will the program display when run as currently written? 10

(b) (4 points) Time for you to take over. Complete the version below so that it will function as described.

```
void add5(                                ) {
```

Solution:

```
void add5(int* x) {
  *x = *x + 5;
}
```

```
}
```

(c) (2 points) Briefly, explain why Prof. Summet's original version did not work the way she wanted it to.

Solution: Professor Summet reassigned the parameter variable when she did `x = &y`. Since C is a pass-by-value language, changes made to the parameter variable don't persist after the function ends. Since she changed the value of the parameter from its original address to that of y, the change doesn't persist after the function ends and the parameter variable's memory space is relinquished.

6. (10 points) Write a function named `replace_zeros` which takes two arguments: an array of integers and the number of elements in the array. Return a **new** version of the given array where each zero value in the array is replaced by number directly to the right of the zero in the array if that number is odd. If there is no odd value to the right of the zero, leave the zero as a zero.

Examples:
`replace_zeros([0, 5, 0, 3], 4)` returns `[5, 5, 3, 3]`

`replace_zeros([0, 4, 0, 3], 4)` returns `[0, 4, 3, 3]`

`replace_zeros([0, 1, 0], 3)` returns `[1, 1, 0]`

Solution: Solutions vary. A sample answer:

```
int[] replace_zeros(int array[], int length) {
  int new_array[length];

  int i;
  for(i = 0; i < length - 1; i++) {
    if (array[i] == 0) {
      if(array[i+1] % 2 != 0) { //number to right is odd
         new_array[i] = array[i+1];
      } else {
        new_array[i] = array[i];
      }
    } else {
      new_array[i] = array[i];
    }

  }
  new_array[i] = array[i]; //copy last element w/o checking to right
  return new_array;
}
```

Scoring:
+2: function header (all parts)
+1: create new array
+1: loop over array
+1: correctly check for 0 and replace w/ number to right if odd
+1: correctly leave 0 if even
+1: leave all other numbers the same +2: deals w/ last element correctly / no buffer overflows
+1: returns array

Common errors: not bounds checking and creating a buffer overflow when checking (unnecessarily) to the right of the last element. Not creating and returning a new array but modifying the parameter instead.

Reference Material

Excess-127 Encoding

Bit Pattern	Value Encoded
00000000	-127
00000001	-126
...	...
01111111	0
10000000	1
10000001	2
...	...
11111111	128

Fractions and decimal equivalents

Fraction	Decimal Value
$\frac{1}{2}$	.5
$\frac{1}{4}$	.25
$\frac{1}{8}$	.125
$\frac{1}{16}$	.0625
$\frac{1}{32}$	.03125

`printf` format strings:

Syntax	Datatype
%i, %d	integer
%f	double, float
%c	char
%s	string
%x, %X	hex rep.
%p	pointer

ASCII chart

Dec	Hex	Char
000	00	(nul)
001	01	(soh)
002	02	(stx)
003	03	(etx)
004	04	(eot)
005	05	(enq)
006	06	(ack)
007	07	(bel)
008	08	(bs)
009	09	(tab)
010	0A	(lf)
011	0B	(vt)
012	0C	(np)
013	0D	(cr)
014	0E	(so)
015	0F	(si)
016	10	(dle)
017	11	(dc1)
018	12	(dc2)
019	13	(dc3)
020	14	(dc4)
021	15	(nak)
022	16	(syn)
023	17	(etb)
024	18	(can)
025	19	(em)
026	1A	(eof)
027	1B	(esc)
028	1C	(fs)
029	1D	(gs)
030	1E	(rs)
031	1F	(us)

Dec	Hex	Char
032	20	␣
033	21	!
034	22	"
035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	-
046	2E	.
047	2F	/
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?

Dec	Hex	Char
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	ˆ
095	5F	_

Dec	Hex	Char
096	60	‘
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	˜
127	7F	DEL