Full Name:______________________________________

Quiz I, Spring 2019 Date: 3/5

Instructions:

- Quiz I takes 70 minutes. Read through all the problems and complete the easy ones first.
- This exam is **closed book**, except that you may bring a single doube-sided page of prepared note.

1 (xx/30)	**2 (xx/25)**	**3 (xx/25)**	**4 (xx/20)**	**Bonus-1 (xx/5)**	**Bonus-2 (xx/5)**	**Total (xx/100)**

This exam assumes 64-bit x86 hardware (little Endian) unless otherwise mentioned.

1 Machine representation, bitwise operation (30 points, 5 points each):

Answer the following multiple-choice questions. Circle *all* answers that apply.

A. Which of the following values is the closest to 1 billion?

1. `1<<10`
2. `1<<20`
3. `1<<30`
4. `0x20000000`
5. `0x40000000`
6. `0x80000000`

B. Which of the following statements are true about IEEE floating point representation and operations?

1. a + b is always equal to b + a
2. (a + b) + c is always equal to a + (b + c)
3. The largest value of 32-bit floating point is larger than the largest value of 64-bit unsigned integer.
4. One can use a 32-bit floating point number to count the total number of Facebook users precisely (assuming Facebook has 2 billion users).

C. Which of the following expression clears the least significant bit of the unsigned int variable `x` while leaving other bits unchanged?

1. `(x >> 1) << 1`
2. `x & 0x7fffffff`
3. `x & 0xefffffff`
4. `x & 0xfffffff7`
5. `x & 0xfffffffe`
6. `x | 0x7fffffff`
7. `x | 0xfffffffe`
8. None of the above

D. Which of the following expression evaluates to 0 *if and only if* the value of int variable `x` is 0?

1. `x & x`
2. `x & 0x00000000`
3. `x | 0xffffffff`
4. `x | 0x00000000`
5. None of the above

E. What is the output of the code snippet below (running on a Little-Endian machine)?

```
long long x = -2;
int *y;
y = (int *)&x;
printf("%d %d\n", y[0], y[1]);
```

1. -1 -1
2. -2 -2
3. -1 -2
4. -2 -1
5. Segmentation fault
6. None of the above

F. What is the output of the code snippet below (running on a Little-Endian machine)?

```
float f = -16.0;
char *p;
p = (char *)&f;
printf("%d\n", *p);
```

1. 16
2. -16
3. 0
4. some positive number
5. some negative number

2 Basic C (25 points, 5 points each)

Answer the following multiple-choice questions. Circle *all* answers that apply.

A. Given variable declaration `int **p;` what is the type of the expression `*p`?

1. `int**`
2. `int*`
3. `int`
4. `void*`
5. None of the above

B. Given variable declaration `int *p;` what is the type of the expression `&p`?

1. `int**`
2. `int*`
3. `int`
4. `void*`
5. None of the above

C. What is the output of the code snippet below (running on a Little-Endian machine)?

```
void foo(int *p) {
   p++;
   (*p)++;
}
int main() {
   int a[3] = {1, 2, 3};
   int *p;
   p = a;
   foo(p);
   foo(p);
   printf("%d %d %d\n", a[0], a[1], a[2]);
}
```

1. 1 4 3
2. 1 3 3
3. 2 3 3
4. 1 3 4
5. 1 2 3
6. None of the above

D. What's the value of variable `p` after executing the statement `char p = '1' - 1;`?

1. `'0'`
2. `0x30`
3. `0x31`
4. `'1'`
5. `'\0'`
6. `0x0`
7. None of the above

E. What is the output of running the following code snippet?

```
char a[5] = {'1', '1', '\0', '1', '\0'};
for (int i = 0; i < 5; i++) {
   printf("[%s]\n", a+i);
}
```

Answer:

3 C MiniLab (25 points):

In Lab1, you are asked to implement a function called `string_token`, to split a string into a sequence of tokens according to a specific delimiter character.

Each call to `string_token` returns a pointer to a null-terminated string containing the next token. A sequence of calls to `string_token` that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. This pointer is saved in the variable pointed to by the `saveptr` argument.

On the first call to `string_token`, `str` should point to the string to be parsed, and the value of `saveptr` is ignored. In subsequent calls, `str` should be NULL, and `saveptr` should be unchanged since the previous call.

The code below is Ben Bitdiddle's implementation of `string_token` and his simple test.

```
char *
string_token(char *str, char delim, char **saveptr)
{
    char *s;
    if (str)
        s = str;
    else
        s = *saveptr;

    char *e;
    e = s;
    while ________________________________ {
        if ((*e) == delim) {
            *e = '\0';
            break;
        }
        ____________________;
    }
    *saveptr = e+1;
    if (e > s)
       return s;

    return NULL;
}
int main()
{
    char **saveptr;
    char test_str[100] = "10;11;12";
    char *token;
    token = string_token(test_str, ';', saveptr);
    while (token) {
       printf("[%s]\n", token);
       token = string_token(NULL, ';', saveptr);
    }
    printf("test_str is [%s]\n", test_str);
}
```

(a) (5 points) Assuming Ben's program is completed and works correctly, what is its expected output? (When answering this question, you can ignore the extra printf statement at line 35)?

(b) (5 points) Please complete line 12 and 17 in Ben's implementation of `string_token` to iterate through the string starting from the location pointed to by `e`.

(c) (5 points) When Ben actually runs his program, the dreaded "Segmentation fault" occurs. When he fires up `gdb`, he sees that the segmentation fault occurs at line 19. What is the reason for the segmentation fault?

1. It's because the type of the right handside expression `e+1` does not match that of the left handside `*saveptr` at line 19.
2. It's because `e+1` points to a location outside of the bound of the string `str`.
3. It's because line 19 is attempting to dereference an illegal address with the expression `*saveptr`.
4. It's because it's a compilation error to write `*saveptr`.

(d) (5 points) Please fix Ben's program to elimininate the segmentation fault. You may only modify code in the `main` function and not elsewhere. (You can directly edit the code in the previous page)

(e) (5 points) What is the output of line 35?

(f) **Bonus question I: (5 points)** Suppose we replace line 28 of Ben's `main` program with the following two lines:

```
int x = 0x623b61;
char *test_str = (char *)&x;
```

Assume Ben's program has been fixed as done in (d). What is the output of the program when running on a little-endian machine (You may ignore the printf at line 35)?

What is the output of the program when when running on a big-endian machine (You may ignore the printf at line 35)?

4 More about C: (20 points)

Ben Bitdiddle is asked to implement `char *int2str(unsigned int n)` which converts an unsigned integer to a null-terminated C string containing the *decimal* representation of the number (You can think of `int2str` as the inverse of `atoi`).

```
1:char *
2:int2str(unsigned int n)
3:{
4:   char str[LEN] = "";
5:   while (n > 0) {
6:
7:     char c = ____________________________;
8:
9:     //insert_front inserts "c" as the 1st character in an existing null-terminated
10:    //string str if new string's length (including null byte) does not exceed LEN.
11:    //Existing characters of str are shifted towards the right to make space for c.
12:    insert_front(c, str, LEN);
13:
14:    n = n/10;
15:  }
16:  return str;
17:}
```

(a) (5 points) What is the minimal value of LEN (i.e. length of the character array `str` storing the returned string of `int2str`)? (The length should include the null-terminating byte).

(b) (5 points) Please complete Ben's implementation of `int2str` by filling out line 7.

Ben wrote the following `main` function to test his implementation of `int2str`.

```
int
main()
{
   unsigned int numbers[2] = {0xff, 11};
   char *strings[2];
   for (int i = 0; i < 2; i++) {
      strings[i] = int2str(numbers[i]);
   }
   for (int i = 0; i < 2; i++) {
      printf("[%s]\n", strings[i]);
   }
}
```

(c) (5 points) What is `main` function's expected output if `int2str` has been implemented correctly?

(d) (5 points) Does Ben's program produce the expected output? If not, please help Ben fix the bug by changing `int2str` function. You should assume that `insert_front` has been implemented correctly. You are not supposed to change the `main` function.

(e) **Bonus question II (5 points):** Please implement the `insert_front` helper function. (Note that we will grade you on both style and correctness. If we cannot figure out what your program does in 5 minutes, we will not give you any points even if your code is correct).

—END of Quiz I—-

Appendix: ASCII

ASCII(7) Linux Programmer’s Manual ASCII(7)

NAME

The following table contains the 128 ASCII characters encoded in octal, decimal, and hexadecimal

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL ’\0’	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL ’\a’ (bell)	107	71	47	G
010	8	08	BS ’\b’ (backspace)	110	72	48	H
011	9	09	HT ’\t’ (horizontal tab)	111	73	49	I
012	10	0A	LF ’\n’ (new line)	112	74	4A	J
013	11	0B	VT ’\v’ (vertical tab)	113	75	4B	K
014	12	0C	FF ’\f’ (form feed)	114	76	4C	L
015	13	0D	CR ’\r’ (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ ’\\’
035	29	1D	GS (group separator)	135	93	5D	]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27		147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL