mt. Code=5 Digipen login:_____

1. **Problem** (4 * 2 pts):

   Given these declarations, what is the value of each of the expressions? Assume logical shift.

```
unsigned short a = 12, b = 15, c = 37;
```

A) 12 B) 15 C) 29 D) 14 E) 26 F) 5 G) none of the listed H) 30 I) 37 J) 13 K) 28 L) 16	1-1._____ `b | a` 1-2._____ `c & b` 1-3._____ `c & b | a` 1-4._____ `c & b | a << 1`

2. **Problem** (5 * 2 pts):

   Convert C declaration into English

   A) an array of 5 pointers to pointers to int

   B) a pointer to a pointer to an array of 5 int

   C) a pointer to a pointer to a function that returns an int

   D) a function that takes an int and returns a pointer to a pointer to an int

   E) an array of 5 pointers to functions taking an int and returning an int

   F) a function that takes an int and returns a pointer to an array of 5 ints

   G) a pointer to an array of 5 pointers to functions that return an int

   H) legal declaration, but not in the list

   I) a pointer to an array of 5 pointers to int

   J) a function that takes an int and returns a pointer to an array of 5 pointers to int.

   K) pointer to a function taking a pointer to function taking an array of 5 integers and returning an int and returning an int

   L) illegal declaration

   M) a pointer to a function that takes an int and returns an int

   N) function taking a pointer to array of 5 ints and returning a pointer to an int

   O) pointer to a function taking a pointer to array of 5 ints and returning an int

   P) a pointer to an array of 5 pointers to functions taking an int and returning an int

   2-1._____ `int (*foo[5])(int);`

   2-2._____ `int *(foo[5])(int);`

   2-3._____ `int *(*foo)[5];`

   2-4._____ `int (*foo)(int (*)(int [5]));`

   2-5._____ `int (*foo)(int(*)[5]);`

3. **Problem** (3 * 2 pts):

You have a SORTED singly-linked list of integers, initially empty. You are about to insert n elements (in inscreasing order). The following questions deal with the number of comparisons that are required to insert n elements. "Comparison" is data comparison - that is we only count comparisons of two data fields, comparisons of pointers (for example to NULL) are NOT counted. Hint: when inserting into an empty list the number of comparisons is 0.

A) $n(n-1)/2$

B) $n-1$

C) use doubly-linked list

D) $(n-1)^2$

E) use local reference strategy

F) use tail pointer

G) $(n-1)^3$

H) use dummy strategy

3-1._____ what is the number of comparisons if inserted numbers are $1, 2, 3, \ldots, n$ – in this order

3-2._____ what is the number of comparisons if inserted numbers are $n, n-1, n-2, \ldots, 2, 1$ – in this order

3-3._____ Suppose we know before hand that numbers (data) are always inserted in the end of the list. How can we use this information to write more efficient code?

4. **Problem** (10 * 1 pts):

For each of the following expressions determine whether it's legal.
For legal expression write down its value.
Assume non-cumulative execution, i.e. all modifications from previous lines ARE LOST.
Note that the value of assignment is value of the right-hand side, if the assignment is legal.

```
short array[] = {3, 6, 2, 4, 7, 8};
/* assume array = 1000, sizeof(int)=sizeof(pointer)=4, sizeof(short)=2 */
short *p1 = &array[1];
short *p3 = &array[3];
short *p5 = &array[5];
```

A) illegal B) legal, but not listed C) 2 D) 4 E) -4 F) 6 G) 7 H) 8 I) 1003 J) 1004 K) 1008 L) 5 M) 1001 N) 10 O) 3 P) 1006 Q) 1000 R) 1 S) 1002	4-1._____ `p1-p5` 4-2._____ `p1-p3+p5` 4-3._____ `p5-2-p1` 4-4._____ `p1 += p5-p3` 4-5._____ `p1 -= p5-p3` 4-6._____ `p5 + p1` 4-7._____ `p5 - p1--` 4-8._____ `*--p3 = 2` 4-9._____ `*(p3+2) = 6` 4-10._____ `p3+2 = &array[2]`

5. **Problem** (4 * 1 pts):

Given the following declaration,

```
short sha[]={1,2,3,4,5,6,7,8};
short* ps1 = sha+1;
short* ps3 = &sha[3];
int*   pi0 = (int*)sha;
int*   pi2 = (int*)(&sha[2]);
```

evaluate expressions. Assume sizeof(int)=4, sizeof(short)=2.

A) 8 B) 16 C) -1 D) -2 E) -6 F) 1 G) illegal H) 3 I) -4 J) 6 K) 2 L) 4	5-1._____ `sizeof(sha)` 5-2._____ `sizeof(*sha)` 5-3._____ `pi2 - pi0` 5-4._____ `pi2 - ps1`

6. **Problem** (7 * 1 pts):

Given the following declaration,

```
int f1(void) { return 16;  }
int f2(void) { return 32;  }
int f3(void) { return 64;  }

int (*pf[])(void) = {f1, f2, f3};
```

evaluate expressions.

A) address of array B) 16 C) 32 D) illegal E) address of f1 F) address of f2 G) 64 H) address of f3	6-1._____ `(*(pf+1))()` 6-2._____ `(*(pf+1))` 6-3._____ `(*pf+1)()` 6-4._____ `pf()` 6-5._____ `pf[1]()` 6-6._____ `(**(pf+1))()` 6-7._____ `f3`

7. **Problem** (8 * 1 pts):

Which of the following implicit casts are (is) legal? Assume Foo is a well-defined struct. Unless declaration is provided with a question, assume

```
Foo f;
const Foo cf;
Foo* p_f;
const Foo* p_cf;
Foo* const cp_f;
```

A) legal B) illegal	7-1._____ `const Foo * p_cf2 = &f;` 7-2._____ `const Foo * p_cf2 = &cf;` 7-3._____ `f = cf;` 7-4._____ `cf = f;` 7-5._____ `p_f = p_cf;` 7-6._____ `p_cf = p_f;` 7-7._____ `p_f = cp_f;` 7-8._____ `cp_f = p_f;`

8. **Problem** (4 * 1 pts):

Given the following declarations, evaluate expressions - use decimal system and sizeof(int)=sizeof(pointer)=4.

```
int array[][4] = { {1,2,3,4}, {11,12,13,14}, {21,22,23,24}, {31,32,33,34} };
int **pp_int = (int**)array;
/* assume array is 1000 */
```

A) garbage B) 1060 C) 1009 D) 1000 E) 1048 F) 1036 G) 1012 H) 1002	8-1._____ `&array[3];` 8-2._____ `&array[3][3];` 8-3._____ `&pp_int[3];` 8-4._____ `pp_int[3][3]`

9. **Problem** (6 pts):

Write a function that accepts an integer and sets its bits 0,2,4 to 1 and bits 1,3 to 0. Points are awarded for correctness, meaningfullness, and compactness of your code.

10. **Problem** (12 pts):

Write a function (provide signature and implementation) that inserts a given node right BEFORE the last node of a singly linked list. If the list is empty – then the new node becomes the only node in the list.

Points are awarded for correctness and compactness of code.

```
struct Node {
  ... /*some data*/
  struct Node *next;
};
```

11. **Problem** (8 pts):

Write code fragment that flips n^{th} bit of an integer i, points are awarded for correctness, efficiency, and compactness. Flipping means if n^{th} bit was originally 0, it becomes 1, and if it was 1, it becomes 0. All other bits should retain their old values.

12. **Problem** (8 pts):

Write the C declaration for each of the English statements below. If a statement describes an illegal declaration, then write ILLEGAL.

a. foo is an array of 7 pointers to int

b. foo is an array of 5 pointers to functions that take an int and returns an int

c. foo is a function that takes a pointer to (an array of 5 integers) and returns a pointer to an array of 2 ints

d. foo is a function that takes a pointer to (a function that takes nothing and returns nothing) and returns a function that (takes an int and returns an int)

13. **Problem** (6 pts):

Study the code below carefully. What does it print out? You should draw a diagram to help visualize the operations. (Hint: The outputted characters form a word.)

```
#include <stdio.h>
void main() {
    char *strs[] = {"In","Ithaca","socialism", "savors","relaxation",0};
    char** pps = strs;
    int i=0;

    while(i<4) {
        printf ("%c", *(*++pps + ++i) - i);
    }
}
```

14. **Problem** (10 pts):

Write a function that accepts an integer *dim* and allocates a 2-D array of doubles of size $dim \times dim$. Array elements should be initialized using formula $i/(j+1)$ for position (i, j).

Function should return the array to a client who is using the code below.

Implement a function that properly deletes array allocated by `create2Darray`.

```
int main() {
  int size = 5;
  ..... a = create2Darray( size );
  if (a) {
    printf("a[%i,%i]=%f",3,2,a[3][2]);
    clean_up( a, size );
  }
  return 0;
}
/* output of this program should be "1" */
```

Incomplete prototype - fill in the blanks and implement:

```
__________ create2Darray ( int dim ) {
```

```
void clean_up ( __________ ) {
```

()	Grouping	exp	N/A
()	Function call	rexp	L-R
[]	Subscript	lexp	L-R
.	Structure member	lexp	L-R
->	Structure pointer member	lexp	L-R
++	Postfix increment	rexp	L-R
--	Postfix decrement	rexp	L-R
!	Logical negate	rexp	R-L
~	One's complement	rexp	R-L
+	Unary plus	rexp	R-L
-	Unary minus	rexp	R-L
++	Prefix increment	rexp	R-L
--	Prefix decrement	rexp	R-L
*	Indirection (dereference)	lexp	R-L
&	Address of	rexp	R-L
sizeof	Size in bytes	rexp	R-L
(type)	Type conversion (cast)	rexp	R-L
*	Multiplication	rexp	L-R
/	Division	rexp	L-R
%	Integer remainder (modulo)	rexp	L-R
+	Addition	rexp	L-R
-	Subtraction	rexp	L-R
<<	Left shift	rexp	L-R
>>	Right shift	rexp	L-R
>	Greater than	rexp	L-R
>=	Greater than or equal	rexp	L-R
<	Less than	rexp	L-R
<=	Less than or equal	rexp	L-R
==	Equal to	rexp	L-R
!=	Not equal to	rexp	L-R
&	Bitwise AND	rexp	L-R
^	Bitwise exclusive OR	rexp	L-R
|	Bitwise inclusive OR	rexp	L-R
&&	Logical AND	rexp	L-R
||	Logical OR	rexp	L-R
?:	Conditional	rexp	N/A
=	Assignment	rexp	R-L
+=	Add to	rexp	R-L
-=	Subtract from	rexp	R-L
*=	Multiply by	rexp	R-L
/=	Divide by	rexp	R-L
%=	Modulo by	rexp	R-L
<<=	Shift left by	rexp	R-L
>>=	Shift right by	rexp	R-L
&=	AND with	rexp	R-L
^=	Exclusive OR with	rexp	R-L
|=	Inclusive OR with	rexp	R-L
,	Comma	rexp	L-R