Problem 1 (2 parts, 32 points) **Storage Allocation and Pointers**

Part A (16 points) Assuming a **64-bit system with 64-bit memory interface and 64-bit addresses**, show how the following global variables map into static memory. Assume they are allocated starting at address 4000 and are properly aligned. **For each variable, draw a box showing its size and position** in the double word memory shown below in which byte addresses increment from left to right. **Label the box with the variable name.** Label each element of an array (e.g., M[0]). Note that int and float are still 32-bits.

```
char   N[] = "Ann";
double F = 17.75;
int     i = 0;
char   *p = N;
int    *q = &i;
int     x = 10;
```

4000	N[0]	N[1]	N[2]	N[3]	s	s	s	s
4008	F	F	F	F	F	F	F	F
4016	i	i	i	i	s	s	s	s
4024	p	p	p	p	p	p	p	p
4032	q	q	q	q	q	q	q	q
4040	x	x	x	x				
4048								
4056								

Part B (16 points) For this part, assume a **32-bit** system, such as MIPS-32.

```
int a = 3;
int b = 5;
char N[] = "Hey!";
int *p = &a;
char *s = N;
p++;
++s;
printf("%d\n", *(p-1);
printf("%c\n", N[3]);
printf("%c\n", *(s+1));
printf("%c\n", *(N+2));
```

Question:	Answer:
How much space (in bytes) is allocated for `p`?	**4 bytes**
How much space (in bytes) is allocated for `s`?	**4 bytes**
What is printed by this statement? `printf("%d\n", *(p-1);`	**3**
What is printed by this statement? `printf("%c\n", N[3]);`	**'!'**
What is printed by this statement? `printf("%c\n", *(s+1));`	**'y'**
What is printed by this statement? `printf("%c\n", *(N+2));`	**'y'**

Problem 2 (2 parts, 28 points) **Parameter Passing**

Part A (20 points) Consider the following C code fragment.

```
typedef struct {
   int height;
   int width;
} rectangle;

int ComputeArea(int L) {
   int       A = 3;
   int       Scales[] = {2, 4};
   rectangle   R;
   int       ScaleHT(int, rectangle *, int []);
   R.height = 10;
   A = ScaleHT(R.height, &R, Scales);
   return(L+A);
}

int ScaleHT(int h, rectangle *P, int S[]) {
   int         w, area;
   w = S[1]*h;
   P->width = w;
   area = h*w;
   S[0] += 8;
   h++;
   return(h+area);
}
```

For each statement below

from `ScaleHT` (as called from ComputeArea), list the resulting value. If the result is an address, just list "address". Also determine if it changes any of `ScaleHT` activation frame variables, `ComputeArea`'s activation frame variables, or both.

Statement in `ScaleHT`	Result (*assigned value*)	`ComputeArea`'s AF variables changed?		`ScaleHT`'s AF variables changed?	
`w = S[1]*h;`	**40**	~~Yes~~	No	Yes	~~No~~
`P->width = w;`	**40**	Yes	~~No~~	~~Yes~~	No
`area = h*w;`	**400**	~~Yes~~	No	Yes	~~No~~
`S[0] += 8;`	**10**	Yes	~~No~~	~~Yes~~	No
`h++;`	**11**	~~Yes~~	No	Yes	~~No~~

Part B (8 points) Consider the MIPS code on the left which implements the array declaration and access on the right, where the variables **Z**, **Y**, **X**, and **Value** reside in $4, $5, $6, and $7 respectively.

`sll $1, $4, 6`
`sll $2, $5, 4`
`add $1, $1, $2`
`add $1, $1, $6`
`sll $1, $1, 2`
`sw $7, Array($1)`
	`int Z, Y, X, Value;`
	…
	
	`int Array[__?______][__4______][__16_____];`
	…
	`Array[Z][Y][X] = Value;`

What does this code reveal about the dimensions of Array? Fill in the blanks in the array declaration with the size of each dimension that can be determined from the code. If a dimension cannot be known from this code, put a "?" in its blank. Assume a 32-bit operating system.

Problem 3 (2 parts, 40 points) **Activation Frames**

Consider the following C code fragment:

```
int ComputeSQ(int Max) {
   int         Sum = 0;
   int         Sqs[3];
   int         M[] = {2, 3, 4};
   int         i;
   for(i=0; i<Max; i++){
     SoS(M[i], &Sum, Sqs, i);
   }
   return(Sum);
}

int SoS(int side, int *Total, int S[], int j) {
   int         square;
   square = side*side;
   *Total += square;
   S[j] = square;
   return(square);
}
```

Part A (18 points) Suppose `ComputeSq` has been called with input `Max`=3 and it calls `SoS` 3 times in its `for` loop. Describe the state of the stack at the end of the *first* execution of `SoS`, <u>just before</u> `SoS` deallocates locals and returns to `ComputeSq` for the first time. Fill in the unshaded boxes to show `ComputeSQ's(CSQ's)` and `SoS's` activation frames. Include a symbolic description and the actual value (in decimal) if it has been assigned. For return addresses, show only the symbolic description; do not include a value. *Indicate the value of the frame pointer ($fp) and stack pointer($sp) at this point in execution*. Assume a 32-bit system.

	address	description	Value
	9880	**RA of CSQ's caller**	
	9876	**FP of CSQ's caller**	
	9872	**Max**	**3**
SP, ComputeSQ's FP	9868	**RV**	
	9864	**Sum**	~~0~~ **4**
	9860	**Sqs[2]**	
	9856	**Sqs[1]**	
	9852	**Sqs[0]**	**4**
	9848	**M[2]**	**4**
	9844	**M[1]**	**3**
	9840	**M[0]**	**2**
	9836	**i**	**0**
$fp: __**9808**__?	9832	**RA of ComputeSQ**	
$sp: __**9804**__?	9828	**FP**	**9868**
	9824	**side**	**2**
	9820	**Total**	**9864**
	9816	**S**	**9852**
	9812	**j**	**0**
	9808	**RV**	**4**
	9804	**square**	**4**
	9800		

Part B (22 points) Write MIPS code fragments to implement the function `SoS` by following the steps below. *Do not use absolute addresses in your code; instead, access variables relative to the frame pointer.* Assume no input parameters are present in registers (i.e., access all parameters from `SoS`'s activation frame). If you assign a register in one part, you may assume it still has that value in a later part. However, changes to variables must update memory.

First, write code to properly set `SoS`'s frame pointer and to allocate space for `SoS`'s local variables and initialize them if necessary.

label	instruction	Comment
SoS:	addi $30, $29, 0	# set FP
	addi $29, $29, -4	# make space for local

`# square = side*side;`

label	instruction	Comment
	lw $1, 16($30)	# load side
	mult $1, $1	# side*side
	mflo $1	# $1 = side*side
	sw $1, -4($30)	# square = side*side

`# *Total += square;`

label	instruction	Comment
	lw $2, 12($30)	# load Total's address
	lw $3, 0($2)	# deref Total
	add $4, $1, $3	# *Total + square
	sw $4, 0($2)	# write to *Total

`# S[j] = square;`

label	instruction	Comment
	lw $2, 8($30)	# load S's base address
	lw $3, 4($30)	# load j
	sll $3, $3, 2	# scale j by 4
	add $5, $2, $3	# compute addr S[j]
	sw $1, 0($5)	# write square to S[j]

`# return(square); (store return value, deallocate locals, and return)`

label	instruction	Comment
	sw $1, 0($30)	# store RV
	addi $29, $30, 0	# deallocate locals
	jr $31	# return to caller