Memory management in C

1.) List all memory resources used by each of the following statements found inside a C function definition.

i. `char *info = "hello";`
ii. `int numbers[4];`
iii. `char *data = (char *) malloc(1024);`
iv. `char **bigdata = (char **) malloc(1024*sizeof(char *));`

i. One character pointer (`info`) resides on the stack; its contents point to a 6-byte string literal (`"hello"` with a null terminator) in the static data section.
ii. This creates a 4-element, 16-byte integer array on the stack; no pointers are created.
iii. Allocates a pointer on the stack; it points to the start of 1024-character block on the heap.
iv. Allocates a pointer on the stack; it points to the start of 1024-pointer block on the heap.

2.) Imagine you are writing an algorithm that uses a binary tree data structure with nodes defined as below. Since space for all the contents of this data structure was allocated on the heap, you need to de-allocate all of the associated memory; write an implementation of `free_tree` that successfully does this. You may assume that the tree is of reasonably shallow depth.

```
struct treenode {
    struct treenode *left_child;
    struct treenode *right_child;
    char *value;
};

void free_tree(struct treenode *root) {
    if (root != NULL) {
        if (root->value != NULL) free(root->value);
        if (root->left_child != NULL) free_tree(root->left_child);
        if (root->right_child != NULL) free_tree(root->right_child);
        free(root);
    }
}
```

3.) **Part I:** Can you find and comment the memory management bugs below? **Part II:** Fix them to the right.

```
#define LEN 64

int *do_things(int *data) {

    int vector[LEN];

    int *tmp = vector;
    // no sizeof(int), no NULL check
    int *values = (int *) malloc(LEN);
    // no sizeof(int), no NULL check
    int *result = (int *) malloc(LEN);

    // do math, populating result
    // NO FREE
    return result;
}
```

```
#define LEN 64

int *do_things(int *data) {
    int vector[LEN];
    int *tmp = vector;
    int *values = (int *) malloc(LEN*sizeof(int));
    if (values == NULL) return NULL;
    int *result = (int *) malloc(LEN*sizeof(int));
    if (result == NULL) {
        free(values);
        return NULL;
    }
    // do math, populating result
    free(values);
    return result;
}
```

Intro to MIPS

Instruction	Syntax	Example
add	`add  dst, src0, src1`	`add  $s0, $s1, $s2`
add immediate	`addi dst, src0, immediate`	`addi $s0, $s1, 12`
shift left logical	`sll  dst, src, shamt`	`sll  $t0, $s0, 4`
load word	`lw   dst, offset(bAddr)`	`lw   $t0, 4($s0)`
store word	`sw   src, offset(bAddr)`	`sw   $t0, 4($s0)`
branch if not equal	`bne  src0, src1, brAddr`	`bne  $t0, $t1, notEq`
branch if equal	`beq  src0, src1, brAddr`	`beq  $t0, $t1, Eq`
jump unconditional	`j    jumpAddr`	`j    jumpWhenDone`
jump register	`jr   reg`	`jr   $ra`

Translate each of the following C-code snippets into MIPS assembly. Use up to eight instructions for each segment, but limit the instructions used to those listed in the table above.

1. Assume `a` is held in `$s0`, `b` is held in `$s1`, `c` is held in `$s2`, and `z` is held in `$s3`.

```
int a=4, b=5, c=6, z;
z = a+b+c+10;
```

```
addi  $s0, $0, 4
addi  $s1, $0, 5
addi  $s2, $0, 6
add   $s3, $s0, $s1
add   $s3, $s3, $s2
addi  $s3, $s3, 10
```

2. Assume `$s0` holds `p` after `int *p = (int *) malloc(3*sizeof(int))` and `$s1` holds `a`.

```
p[0] = 0;
int a = 2;
p[1] = a;
p[a] = a;
```

```
sw    $0, 0($s0)
addiu $s1, $0, 2
sw    $s1, 4($s0)
sll   $t0, $s1, 2    # multiply by 4
addu  $t1, $t0, $s0
sw    $s1, 0($t1)
```

3. Assume `$s0` holds `a` and `$s1` holds `b`.

```
int a = 5, b = 10;
if (a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
```

```
      addiu $s0, $0, 5
      addiu $s1, $0, 10
      add   $t0, $s0, $s0
      bne   $t0, $s1, else
      add   $s0, $0, $0
      j     exit
else: addiu $s1, $s0, -1
exit: ...                  # done!
```

Interpret the following MIPS assembly code and provide a written explanation of what it does.

```
addi $s0, $0,  0
      addi $s1, $0,  1
      addi $t0, $0,  30
loop: beq  $s0, $t0, done
      sll  $s1, $s1, 1
      addi $s0, $s0, 1
      j    loop
done: # done!
```

This computes the number 2^{30} by iteratively doubling a value starting at one 30 times.

MIPS Control Flow

1.) Translate the following abstract branch conditions to minimal MIPS instruction sequences.

- Branch to `exit` if `$s0 < $s1`

```
slt   $t0, $s0, $s1
bne   $t0, $zero, exit
```

- Branch to `done` if `$s0 <= $s1`

```
slt   $t0, $s1, $s0
beq   $t0, $zero, done
```

- Branch to `next` if `$s0 < 1`

```
slti  $t0, $s0, 1
bne   $t0, $zero, next
```

- Branch to `finish` if `$s0 >= 0`

```
slti  $t0, $s0, 0
beq   $t0, $zero, finish
```

2.) Translate the following `strcpy` implementation to MIPS. Assume `$s1` holds the address of the first byte of the array `s1` and `$s2` holds the contents of the pointer variable `s2`.

```
// strcpy:
// char s1[] = Hello!;
// char *s2 = malloc(sizeof(char)*7);
int i=0;
do {
    s2[i] = s1[i];
    i++;
} while (s1[i] != '\0');
s2[i] = '\0';
```

```
      addi $t0, $0,  0
Loop: add  $t1, $s1, $t0   # s1[i]
      add  $t2, $s2, $t0   # s2[i]
      lb   $t3, 0($t1)     # char is
      sb   $t3, 0($t2)     #   1 byte!
      addi $t0, $t0, 1
      addi $t1, $t1, 1
      lb   $t4, 0($t1)
      beq  $t4, $0, Done
      j    Loop
Done: addi $t2, $t2, 1
      sb   $t4, 0($t2)
```

3.) Translate the following algorithm that finds the N^th Fibonacci number to MIPS assembly. Assume `$s0` holds `N`, `$s1` holds `fib`, `$t0` holds `i`, and `$t1` holds `j`.

```
int fib = 1, i = 1, j = 1;

if (N==0)      return 0;
else if (N==1) return 1;
N -= 2;

while (N != 0) {
    fib = i + j;
    j = i;
    i = fib;
    N--;
}

return fib;
```

```
      beq  $s0, $0,  Ret0
      addi $t2, $0,  1
      beq  $s0, $t2, Ret1
      subi $s0, $s0, 2
Loop: beq  $s0, $0,  RetF
      add $s1, $t0, $t1
      addi $t0, $t1, 0
      addi $t1, $s1, 0
      subi $s0, $s0, 1
      j    Loop
Ret0: addi $v0, $0,  0
      j    Done
Ret1: addi $v0, $0,  1
      j    Done
RetF: add  $v0, $0,  $s1
Done: jr   $ra
```

4.) Assuming `$a0` and `$a1` hold integer pointers, swap the values they point via the stack and return control.

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
addiu $sp, $sp, -4
lw    $t0, 0($a0)
sw    $t0, 0($sp)
lw    $t0, 0($a1)
sw    $t0, 0($a0)
lw    $t0, 0($sp)
sw    $t0, 0($a1)
addiu $sp, $sp, 4
jr    $ra
```

MIPS Pseudoinstructions

MIPS assemblers recognize several *pseudoinstructions* – assembly code statements that are not actually part of the MIPS ISA – that get converted to short seqeunces of instructions. For each of the following pseudoinstructions, give the shortest possible instruction sequence you can find. Note that the register `$at` ("assembler temporary") is reserved for use as a temporary variable by the assembler.

- `not $dst, $src`

```
nor   $dst, $src, $src
```

- `abs $dst, $src`

```
      slt   $at,  $zero, $src
      bne   $at,  $zero, pos
      sub   $dst, $zero, $src
      j     done
pos:  add   $dst, $zero, $src
done: ...
```

- `bgt $src0, $src1, label`

```
slt   $at, $src1, $src0
bne   $at, $zero, label
```

- `li $dst, imm`$_{32}$ `# (32b immediate move)`

```
lui   $dst, imm[31:16]
ori   $dst, $dst, imm[15:0]

# Why doesn't the version below work?
lui   $dst, imm[31:16]
addi  $dst, $dst, imm[15:0]
```

MIPS Machine Code

Instructions are represented as bits, same as everything else! All instructions fit in a word (32 bits). In order to cover all the different instructions, there are 3 different instruction types:

`R-Format`	–	`opcode(6)`	`rs(5)`	`rt(5)`	`rd(5)`	`shamt(5)`	`funct(6)`
`I-Format`	–	`opcode(6)`	`rs(5)`	`rt(5)`	`immediate(16)`		
`J-Format`	–	`opcode(6)`	`target address(26)`				

Here the instruction formats are written out by field name and width in bits in parentheses. The first bit of the opcode is the MSB and the other end is the LSB. The fields are used as follows:

`opcode`	indicates operation, or arithmetic family of operations (for `opcode` 0, which is R-type)
`funct`	indicates specific operation within arithmetic family of operations
`rs`, `rt`, `rd`	for R-type, `rs` and `rt` are sources with `rd` as destination – rules vary for other formats!
`shamt`	shift amount for instructions that perform shifts
`immediate`	Relative address or constant
`address`	Absolute address

Practice problem: Decode the instructions a.) `0x02114824` b.) `0x03E00008` c.) `0x292A0020`

a.) `0x02114824` → `and $9, $16, $17` → `and $t1, $s0, $s1`

b.) `0x03E00008` → `jr $31` → `jr $ra`

c.) `0x02114824` → `slti $10, $9, 32` → `slti $t2, $t1, 32`

Assembling and Linking (credits to Justin Hsia)

How many passes over assembly code does an assembler have to make and why?

Two. First to find all label addresses, second for instruction conversion.

The linker resolves issues in relative or absolute addressing?

Absolute addressing.

What does RISC stand for? How is this related to pseudoinstructions?

Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.

MIPS Addressing

Assuming the first instruction is at address 0x000 (which is really 0b 00 0000 0000 0000), fill in the fields below. Use decimal for immediate and hex for addresses.

```
0x000 beq  $s0, $0,  Ret0          imm = _12__
      addi $t0, $0,  1
      beq  $s0, $t0, Ret1          imm = _12__
      subi $s0, $s0, 2
      addi $s1, $0,  1
      addi $t0, $0,  1
      addi $t1, $0,  1
Loop: beq  $s0, $0,  RetF          imm = __9__
      add $s1, $t0, $t1
      addi $t0, $t1, 0
      addi $t1, $s1, 0
      subi $s0, $s0, 1
      j    Loop                    addr = _0x007____
Ret0: addi $v0, $0,  0
      j    Done                    addr = _0x012____
Ret1: addi $v0, $0,  1
      j    Done                    addr = _0x012____
RetF: add  $v0, $0,  $s1
Done: jr   $ra
```

Which instruction can reach more addresses, `j` or `jr`?

`jr` can reach all 2$^{\wedge 32}$ addresses while `j` can only reach 2$^{\wedge 26+2}$ = 2$^{\wedge 28}$.

What is the maximum jump distance of `j` and `jr`?

`j`: immediate is 0x000000 to 0x3fffff, so 2$^{\wedge 26}$ – 1 words = 2$^{\wedge 28}$ – 4 bytes.

`jr`: 0x0000 0000 to 0xffff fffc, so 2$^{\wedge 32}$ – 4 bytes = 2$^{\wedge 30}$ – 1 instructions.

MIPS Calling Conventions

When calling a function in MIPS, who needs to save the following variables to the stack? Answer **R** for the caller, **E** for the callee, or **N** for neither.

`$v0-$v1` _R_ `$a0-$a3` _R_ `$t0-$t9` _R_ `$s0-$s7` _E_ `$sp` _N_ `$ra` _E_

Now assume our function `foo` calls another function `bar`, which is known to call other functions. `foo` takes one argument and uses `$t0` and `$s0`. `bar` takes two arguments, returns an integer, and uses `$t0-$t1` and `$s0-$s1`.

In the box, draw a possible ordering of the stack just before `bar` calls a function (you may not need all the spaces). Add "(f)" if the register is stored by `foo` and "(b)" if the register is stored by `bar`.

$ra (f)	$v0 (b)
$s0 (f)	$a0 (b)
$v0 (f)	$a1 (b)
$a0 (f)	$t0 (b)
$t0 (f)	$t1 (b)
$ra (b)	
$s0 (b)	
$s1 (b)	

Instruction Conversion Practice

```
# $s0 -> int * (address)
# $a0 -> int
0x000  addi $v0, $0,  0
Loop:  slt  $t0, $v0, $a0
       beq  $t0, $0,  Done
       sll  $t1, $v0, 2
       addu $t2, $s0, $t1
       sw   $t0, 4($t2)
       addi $v0, $v0, 1
       j    Loop
Done:  # done!
```

Line 1:	8 00 1000	0 0 0000	2 0 0010	0 0000 0000 0000 0000		
Line 2:	0 00 0000	2 0 0010	4 0 0100	8 0 1000	0 0 0000	42 10 1010
Line 3:	4 00 0100	8 0 1000	0 0 0000	5 0000 0000 0000 0101		
Line 4:	0 00 0000	0 0 0000	2 0 0010	9 0 1001	2 0 0010	0 00 0000
Line 5:	0 00 0000	16 1 0000	9 0 1001	10 0 1010	0 0 0000	33 10 0001
Line 6:	43 10 1011	10 0 1010	8 0 1000	4 0000 0000 0000 0100		
Line 7:	8 00 1000	2 0 0010	2 0 0010	1 0000 0000 0000 0001		
Line 8:	2 00 0010	1 00 0000 0000 0000 0000 0000 0001				

Floating Point Rounding Modes:

Round the following binary numbers to the nearest **integer** using each of the four modes, when needed:

	0.00_2	0.01_2	0.10_2	0.11_2	1.00_2	1.01_2	1.10_2	1.11_2
Half-way?	No	No	Yes	No	No	No	Yes	No
$+\infty$	0	0	1	1	1	1	10_2	10_2
$-\infty$	0	0	0	1	1	1	1	10_2
Zero	0	0	0	1	1	1	1	10_2
Even	0	0	0	1	1	1	10_2	10_2

Floating Point

Convert the following decimal numbers into binary (not float):

1.5	0.25	0.8	-16.5
$= 2^0+2^{-1}$	$= 2^{-2}$	$= 2^{-1}+2^{-2}+2^{-5}+2^{-6}+\ldots$	$= -(2^4+2^{-1})$
$= 1.1_2$	$= 0.01_2$	$= 0.\overline{1100}_2$ (repeating)	$= -10000.1_2$

Give the best hex representation of the following numbers (using single precision floats):

1.0	-7.5	(1.0/3.0)	(186.334/0.0)
$= 1_2$	$= -111.1_2$	$= 0.\overline{01}_2$	$= +\infty$
S = 0	S = 1	S = 0	S = 0
E = 127	E = 129	E = 125	E = 255
M = 0	M = 0b1110…0	M = $0b\overline{01}$	M = 0
0x3f800000	0xc0f00000	0x3eaaaaaa	0x7f800000

What is the value of the following single precision floats?

0x0	0xff94beef	0x1
0	-NaN	$2^{-126} \times 2^{-23} = 2^{-149}$

These were all special numbers. Remember that exponent for denorm is 2^{-126}.

Performance Metrics

Exercises

You are the lead developer of a new video game at AE, Inc. The graphics are quite sexy, but the frame rates (performance) are horrible. Doubly unfortunately, you have to show it off at a shareholder meeting tomorrow. What do you do?

1) You need to render your latest and greatest über-l33t animation. If your rendering software contains the following mix of instructions, which processor is the best choice?

Operation	Frequency
ALU	30%
Load	30%
Store	20%
Branch	20%

A's CPI	B's CPI	C's CPI
1	1	1
3	5	3
2	3	4
3	2	2

```
Average CPI:

A: 1*.3 + 3*.3 + 2*.2 + 3*.2 = 2.2
B: 2.8
C: 2.4

A wins.
```

2) What if the processors had different clock speeds? Assume A is a 1 Ghz processor, B is a 1.5 Ghz processor, and C is a 750 Mhz processor.

```
1/frequency = seconds/cycle
cycles/inst * seconds/cycle = seconds/inst, a better estimate of
performance.

seconds / cycle: A = 1 ns, B = .66 ns, C = 1.33 ns
seconds / inst: A = 2.2 ns, B = 1.86 ns, C = 3.2 ns.

B wins.
```

3) But wait, these processors are made by different manufacturers, and use different instruction sets. So the renderer (for the different architectures) takes a different number of instructions on each. Which is best if your main loop on A averages 1000 instructions; on B it averages 800 instructions; and on C it averages 1200 instructions?

```
We now have instructions/program.
seconds/inst [from part b] * inst/program = seconds/program, the
runtime.

instructions / program: A = 1000, B = 800, C = 1200
seconds/program: A = 2.2 us, B = 1.493 us, C = 3.84 us.
B produces the fastest program, and wins.
```