CS 304 Homework 4 Sept. 24, 2013

Due: 11:00am, Tuesday, October 1 in class

1. Convert this function into pointer-based code.

```
void shift(int a[], int n) {
   int i;
   for(i = 0; i != n-1; i++)
      a[i] = a[i+1];
}
```

2. Pointers

On the left is a short C program (blocks.c) that uses a series of operations involving pointers. Fill in the blanks on the right with the value of the requested variable AFTER the execution of the instruction across from it (use char notation for characters and hex for addresses). Assume the address of the blocks array is 0x4680.

Note: Make sure you do this by hand at first. The point here is to learn how pointers and pointer statements work.

```
int main(void) {
   char blocks[3] = {'A','B','C'};    blocks = 0x4680
   char *ptr = &blocks[0];            ptr = _____
   char temp;                         temp = _____

   temp = blocks[0];                  temp = _____
   temp = *(blocks + 2);              temp = _____
   temp = *(ptr + 1);                 temp = _____
   temp = *ptr;                       temp = _____

   ptr = blocks + 2;                  ptr = _____
   temp = *ptr;                       temp = _____
   temp = *(ptr - 1);                 temp = _____

   ptr = blocks;                      ptr = _____
   temp = *++ptr;                     ptr = _____, temp = _____
   temp = ++*ptr;                     ptr = _____, temp = _____
   temp = *ptr++;                     ptr = _____, temp = _____
   temp = *ptr;                       temp = _____

   return 0;
}
```

3. Complete the following setName, getStudentID, and setStudentID functions. You may assume the pointers given are valid and not null.

```
#define MAX_NAME_LEN 127

typedef struct {
        char name[MAX_NAME_LEN + 1];
        unsigned long sid;
} Student;

/* return the name of student s */
const char* getName (const Student* s) {
        return s->name;
}

/* set the name of student s
If name is too long, cut off characters after the maximum number of characters allowed.
*/
void setName(Student* s, const char* name) {









}

/* return the SID of student s */
unsigned long getStudentID(const Student* s) {









}

/* set the SID of student s */
void setStudentID(Student* s, unsigned long sid) {









}
```

4. What is the logical error in the following function?

```
Student* makeAndrew(void) {
        Student s;
        setName(&s, "Andrew");
        setStudentID(&s, 12345678);
        return &s;
}
```

5. This problem tests your understanding of stack frames. It is based on the following C function:

```
#include <stdio.h>
char buf2[1024];
int temp2;
char buf3[512];
int temp3;

int proc(int a, int b, int c)
{
   int temp1;
   char buf1[2048];
   int temp4;
   temp1 = 7*c + (a<<4);
   gets(buf1);
   temp2 = 11*temp1 + c*31;
   gets(buf2);
   temp3 = 64* (c + 4* temp1 + 8*temp2);
   temp4 = (15* temp2) + (temp1<<5);
   gets(buf3);
   printf("%s %s\n",buf1,buf2);
   return 12*temp1+11*temp2+10*temp3+9*temp4;
}
```

This yields the following machine code:

```
Dump of assembler code for function proc:
0x08048408 <proc+0>:    push   %ebp
0x08048409 <proc+1>:    mov    %esp,%ebp
0x0804840b <proc+3>:    push   %edi
0x0804840c <proc+4>:    push   %esi
0x0804840d <proc+5>:    push   %ebx
0x0804840e <proc+6>:    sub    $0x818,%esp
0x08048414 <proc+12>:   mov    0x10(%ebp),%esi
0x08048417 <proc+15>:   lea    0x0(,%esi,8),%ebx
0x0804841e <proc+22>:   sub    %esi,%ebx
0x08048420 <proc+24>:   mov    0x8(%ebp),%eax
0x08048423 <proc+27>:   shl    $0x4,%eax
0x08048426 <proc+30>:   add    %eax,%ebx
0x08048428 <proc+32>:   lea    0xfffff7f4(%ebp),%eax
0x0804842e <proc+38>:   push   %eax
0x0804842f <proc+39>:   call   0x8048308 <gets>
0x08048434 <proc+44>:   lea    (%ebx,%ebx,4),%edx
0x08048437 <proc+47>:   lea    (%ebx,%edx,2),%edx
0x0804843a <proc+50>:   mov    %esi,%eax
0x0804843c <proc+52>:   shl    $0x5,%eax
0x0804843f <proc+55>:   sub    %esi,%eax
0x08048441 <proc+57>:   add    %eax,%edx
0x08048443 <proc+59>:   mov    %edx,0x8049940
0x08048449 <proc+65>:   movl   $0x8049960,(%esp)
0x08048450 <proc+72>:   call   0x8048308 <gets>
0x08048455 <proc+77>:   mov    0x8049940,%edi
0x0804845b <proc+83>:   lea    (%ebx,%edi,2),%eax
0x0804845e <proc+86>:   lea    (%esi,%eax,4),%eax
0x08048461 <proc+89>:   shl    $0x6,%eax
0x08048464 <proc+92>:   mov    %eax,0x8049d60
0x08048469 <proc+97>:   movl   $0x8049740,(%esp)
      (continued above right)
```

```
      (continued from below left)
0x08048470 <proc+104>:  call   0x8048308 <gets>
0x08048475 <proc+109>:  add    $0xc,%esp
0x08048478 <proc+112>:  push   $0x8049960
0x0804847d <proc+117>:  lea    0xfffff7f4(%ebp),%eax
0x08048483 <proc+123>:  push   %eax
0x08048484 <proc+124>:  push   $0x8048604
0x08048489 <proc+129>:  call   0x8048328 <printf>
0x0804848e <proc+134>:  lea    (%ebx,%ebx,2),%eax
0x08048491 <proc+137>:  mov    0x8049940,%edx
0x08048497 <proc+143>:  lea    (%edx,%edx,4),%ecx
0x0804849a <proc+146>:  lea    (%edx,%ecx,2),%ecx
0x0804849d <proc+149>:  lea    (%ecx,%eax,4),%eax
0x080484a0 <proc+152>:  mov    %edi,%ecx
0x080484a2 <proc+154>:  shl    $0x4,%ecx
0x080484a5 <proc+157>:  sub    %edi,%ecx
0x080484a7 <proc+159>:  shl    $0x5,%ebx
0x080484aa <proc+162>:  add    %ebx,%ecx
0x080484ac <proc+164>:  lea    (%ecx,%ecx,8),%ecx
0x080484af <proc+167>:  mov    0x8049d60,%edx
0x080484b5 <proc+173>:  lea    (%edx,%edx,4),%edx
0x080484b8 <proc+176>:  lea    (%ecx,%edx,2),%edx
0x080484bb <proc+179>:  add    %edx,%eax
0x080484bd <proc+181>:  lea    0xfffffff4(%ebp),%esp
0x080484c0 <proc+184>:  pop    %ebx
0x080484c1 <proc+185>:  pop    %esi
0x080484c2 <proc+186>:  pop    %edi
0x080484c3 <proc+187>:  pop    %ebp
0x080484c4 <proc+188>:  ret
End of assembler dump.
```

Give the location of each of the following program variables. If the variable is located on the stack, give the location as an offset from `%ebp`, such as `12(%ebp)` or `%ebp + 12`. If the variable is not located on the stack, then state where the variable is located.

`temp1` _____ `temp4` _____ `buf1` _____

`temp2` _____ `buf2` _____

6. The disassembly on the right was produced by `gdb` from the source code shown on the left.

```
/* CS 304 HW4*/
#include <stdio.h>

int sum( int a[] )
{
   return a[0] + a[1] + a[2];
}

int main(void)
{
   int u[3] = { 55, 44, 33 };
   int v = sum(u);
   printf("%d\n",v);
   return 0;
}
```

```
(gdb) disas sum+0 main+61
Dump of assembler code from 0x80483d8 to 0x8048426:
0x080483d8 <sum+0>:     push   %ebp
0x080483d9 <sum+1>:     mov    %esp,%ebp
0x080483db <sum+3>:     mov    0x8(%ebp),%edx
0x080483de <sum+6>:     mov    (%edx),%eax
0x080483e0 <sum+8>:     add    0x4(%edx),%eax
0x080483e3 <sum+11>:    add    0x8(%edx),%eax
0x080483e6 <sum+14>:    pop    %ebp
0x080483e7 <sum+15>:    ret
0x080483e8 <main+0>:    push   %ebp
0x080483e9 <main+1>:    mov    %esp,%ebp
0x080483eb <main+3>:    sub    $0x18,%esp
0x080483ee <main+6>:    and    $0xfffffff0,%esp
0x080483f1 <main+9>:    sub    $0x18,%esp
0x080483f4 <main+12>:   movl   $0x37,0xfffffff4(%ebp)
0x080483fb <main+19>:   movl   $0x2c,0xfffffff8(%ebp)
0x08048402 <main+26>:   movl   $0x21,0xfffffffc(%ebp)
0x08048409 <main+33>:   lea    0xfffffff4(%ebp),%eax
0x0804840c <main+36>:   push   %eax
0x0804840d <main+37>:   call   0x80483d8 <sum>
0x08048412 <main+42>:   mov    %eax,(%esp)
0x08048415 <main+45>:   push   $0x8048554
0x0804841a <main+50>:   call   0x80482f8 <printf>
0x0804841f <main+55>:   mov    $0x0,%eax
0x08048424 <main+60>:   leave
0x08048425 <main+61>:   ret
End of assembler dump.
```

Suppose that the registers have the following contents on entry to `main` at `0x080483e8`:

| register | contents |
|---|---|
| eax | 0xbf9d1994 |
| ebx | 0x40157ff4 |
| ecx | 0x40051e65 |
| edx | 0x00000001 |
| esp | 0xbf9d191c |
| ebp | 0xbf9d1968 |
| esi | 0x00000000 |
| edi | 0x40015cc0 |
| eip | 0x080483e8 |

Sketch the stack frames for this program at the point **just after** execution of the `mov` instruction in `sum` at location `0x080483d9` (`<sum+1>`). Your stack sketch should contain all memory addresses in the stack and their contents (in a unit of 4 bytes), which are possibly known to you. All stack addresses and stack contents must be displayed in hexadecimal.

The program was compiled with the `-O` flag for optimization.