CAS CS 210 - Computer Systems
Fall 2014

SOLUTIONS: Problem Set 1 (PS1) ('C' Basics, Logic and Data Representaiton)
Out: September 09
Due: Part A September 18, 1:30 pm; Part B September 23, 1:30 pm

NO LATE SUBMISSIONS WILL BE ACCEPTED

To be completed individually. For all questions, show your work in obtaining the final answers.

PART A: 'C' Basics and Logic

READ: K&R: chapter 1, 5.1-5.6, 6.1-6.4

1) hello.c

```
1 #include<stdio.h>
2
3 int
4 main(int argc, char **argv)
5 {
6   printf("Hello World\n");
7   return 0;
8 }
```

Describe the meaning/side effect of each non-blank line.

```
1: #include <stdio.h>
   This is a 'C' preprocessor directive that causes the include file stdio.h,
   that describes the standard input/output (io) library functions, to be inserted
   here.  BONUS: This ensure that the compiler can properly generate calls to
   these functions as from within the current file -- this allows you to
   invoke calls such as printf.

3: int
   Part of the declaration of the main function that indicates that
   it's return type is an integer

4: main(int argc, char **argv)
   continuation of the declaration of the main function.  Specifices
   that main is a function that takes two arguments, first an
   integer referenced as argc and  secondly an array of char
   pointers.  BONUS:  argc number of arguments to the program and
```

```
   argv is an array of the space seperated strings that form the
   command line

5: {
    indicates that the following block of statements defines the body
    of the function main

6: printf("Hello World\n");
   causes the main function to invoke/call the printf routine of the
   standard 'c' library.  This function will print the string "Hello
   World" with a newline to the screen/standard out of the program.
   BONUS: printf is being passed a single char * argument as the
   format string.

7: return 0;
   causes execution of main to terminate by returnig to the caller with
   a return value of 0;

8: }
   terminates the block of statements that defines the body of the
   main function
```

2) Memory and Pointers

```
1  #include <stdio.h>
2
3  int myint;
4  int *ip;
5  char mystring[6] = "hello";
6  char *cp;
7
8  int myfunc0(int x, int *ip)
9  {
10   *ip = *ip + 2;
11   x++;
12   return x;
13 }
14
15 int myfunc1(char *c)
16 {
17   if (*c >= 'a' && *c <= 'z') {
18     *c +='A' - 'a';
19     return 1;
20   }
21   return 0;
22 }
23
24 int main(int argc, char **argv)
25 {
26   myint = 0;
27   ip = &myint;
28   cp = mystring;
29
30   while (myfunc1(cp)) {
31     printf("%d\n", myfunc0(myint, ip));
32     cp++;
33   }
34
35   printf("mystring:%s  myint:%d\n", mystring, myint);
36   return 0;
37 }
```

Please provide the output below for the program listing.

```
1
3
5
7
9
mystring:HELLO  myint:10
```

Please fill in the missing values for the following table assuming that we stop the program just prior to it exiting at line 36. All address values should be written as 8 digit hex values and all integer values as simple decimals.

Name	Address	Value
mystring[0]	0x0804972c	72 'H'
mystring[1]	0x0804972d	69 'E'
mystring[2]	0x0804972e	76 'L'
mystring[3]	0x0804972f	76 'L'
mystring[4]	0x08049730	79 'O'
mystring[5]	0x08049731	0
ip	0x0804973c	0x08049740
myint	0x08049740	10
cp	0x08049744	0x08049731

```
How answers can be validated using gdb:
NOTE: YOU MAY GET DIFFERENT ADDRESS IN WHICH CASE YOU WOULD NEED TO
TRANSLATE TO GET THE RIGHT ANSWERS

csa2$ gcc -m32 -g memory.c -o memmory
csa2$ gdb memory
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/fac1/jappavoo/tmp/cs210/memory...done.
(gdb) list
17   if (*c >= 'a' && *c <= 'z') {
18     *c +='A' - 'a';
19     return 1;
20   }
21   return 0;
22 }
23
24 int main(int argc, char **argv)
```

```
25 {
26   myint = 0;
(gdb) list
27   ip = &myint;
28   cp = mystring;
29
30   while (myfunc1(cp)) {
31     printf("%d\n", myfunc0(myint, ip));
32     cp++;
33   }
34
35   printf("mystring:%s  myint:%d\n", mystring, myint);
36   return 0;
(gdb) b 36
Breakpoint 1 at 0x80484a0: file memory.c, line 36.
(gdb) run
Starting program: /home/fac1/jappavoo/tmp/cs210/memory
1
3
5
7
9
mystring:HELLO  myint:10

Breakpoint 1, main (argc=1, argv=0xffffd424) at memory.c:36
36   return 0;
(gdb) p mystring[0]
$1 = 72 'H'
(gdb) p &mystring[0]
$2 = 0x804972c "HELLO"
(gdb) p &mystring[1]
$3 = 0x804972d "ELLO"
(gdb) p mystring[1]
$4 = 69 'E'
(gdb) p &mystring[2]
$5 = 0x804972e "LLO"
(gdb) p mystring[2]
$6 = 76 'L'
(gdb) p &mystring[3]
$7 = 0x804972f "LO"
(gdb) p mystring[3]
$8 = 76 'L'
(gdb) p mystring[4]
$9 = 79 'O'
(gdb) p &mystring[4]
```

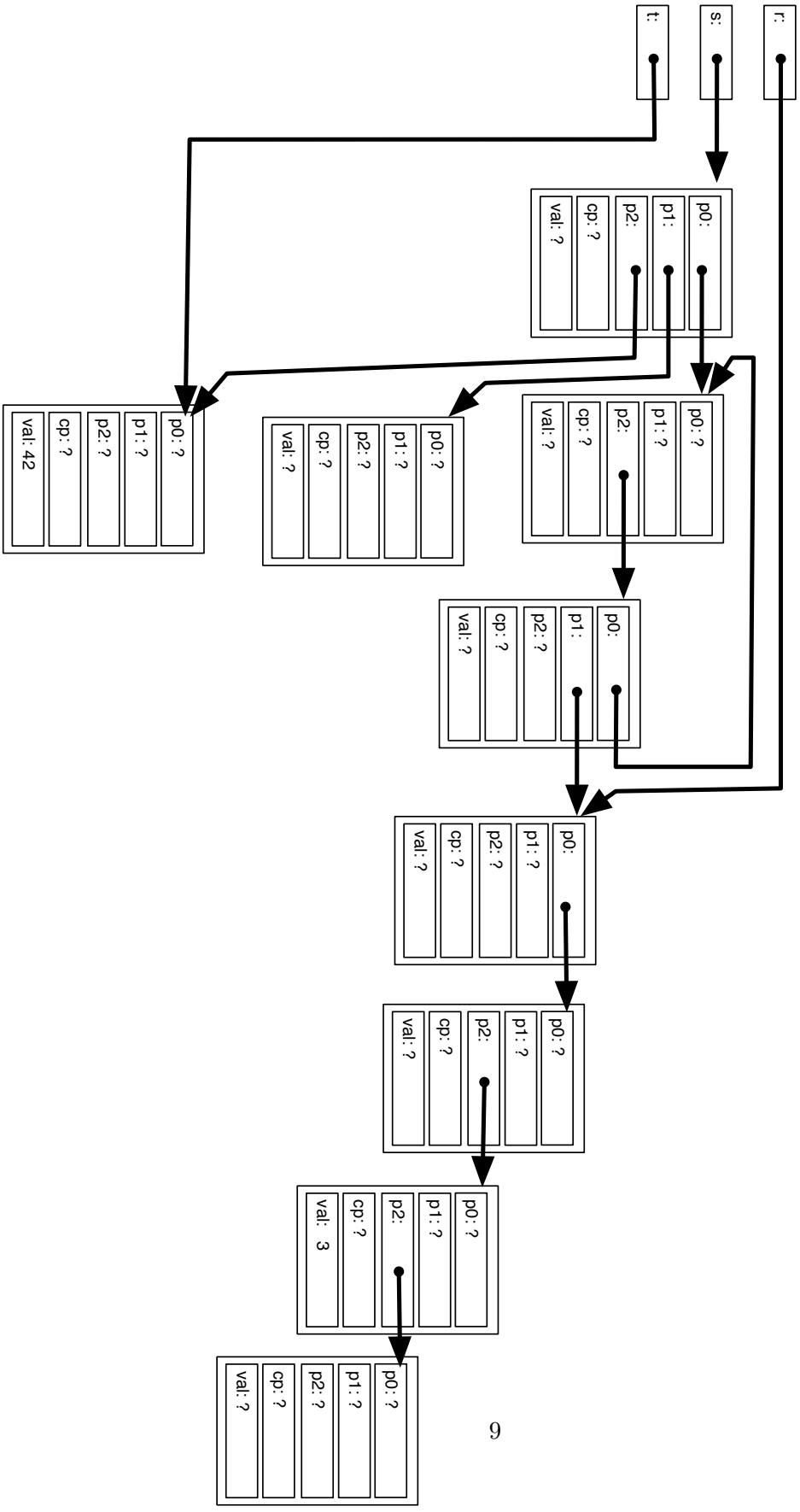
```
$10 = 0x8049730 "O"
(gdb) p &mystring[5]
$11 = 0x8049731 ""
(gdb) p mystring[5]
$12 = 0 '\000'
(gdb) p ip
$13 = (int *) 0x8049740
(gdb) p myint
$14 = 10
(gdb) p cp
$15 = 0x8049731 ""
(gdb)
```

3) Pointers and Structs

```
1  struct myStruct {
2    struct myStruct *p0;
3    struct myStruct *p1;
4    struct myStruct *p2;
5    char            *cp;
6    int              val;
7  };
8
9  struct myStruct *r;
10 struct myStruct *s;
11 struct myStruct *t;
12
13 s = malloc(sizeof(struct myStruct));
14 s->p0 = malloc(sizeof(struct myStruct));
15 s->p0->p2 = malloc(sizeof(struct myStruct));
16 s->p0->p2->p1 = malloc(sizeof(struct myStruct));
17 s->p0->p2->p0 = s->p0;
18 r=s->p0->p2->p1;
19 s->p1 = malloc(sizeof(struct myStruct));
20 r->p0 = malloc(sizeof(struct myStruct));
21 r->p0->p2 = malloc(sizeof(struct myStruct));
22 r->p0->p2->p2 = malloc(sizeof(struct myStruct));
23 t = malloc(size(struct myStruct));
24 s->p0->p2->p1->p0->p2->val = 21;
25 t->val = 42;
26 s->p2 = t;
27 s->p0->p2->p0->p2->p1->p0->p2->val = 3;
```

A struct is a multi-byte programmer defined type that groups together several members (K&R ch 6). Sizeof can be used to determine the aggregate number of bytes that a particular struct type requires. Malloc is a standard 'C' library call that dynamically allocates the requested number of bytes of memory (K&R 7.8.5). Malloc returns the address of the newly allocated memory. Storing the address in a variable of the appropriate pointer type allows you to access the memory allocated by malloc. In the case of a struct pointer you use the '->', called the member selection operator, to access a particular field of the struct pointed too. For futher details on structs and malloc see the appropriate sections in K&R.

Complete the diagram on the next page. Illustrate the side effect of the above code fragment. Draw all additional boxe and complete and add arrows as needed. Note assume there are no failures in calls to malloc. You may use '?' to indicate unknown values of fields. Be sure to indicate all instances of the struct and all field values.


t:
s:
r:
val: ?
cp: ?
p2:
p1:
p0:
val: 42
p0: ?
p1: ?
p2: ?
3

4) Logic Gates and Truth Tables)

1. Prove both of DeMorgan's Laws using Truth Tables. DeMorgan's Laws are:

   (a) Law 1: Stated in english and in 'C'

   English: Not A and B is the same as Not A or Not B
   'C': $!(A\&\&B) == (!A||!B)$

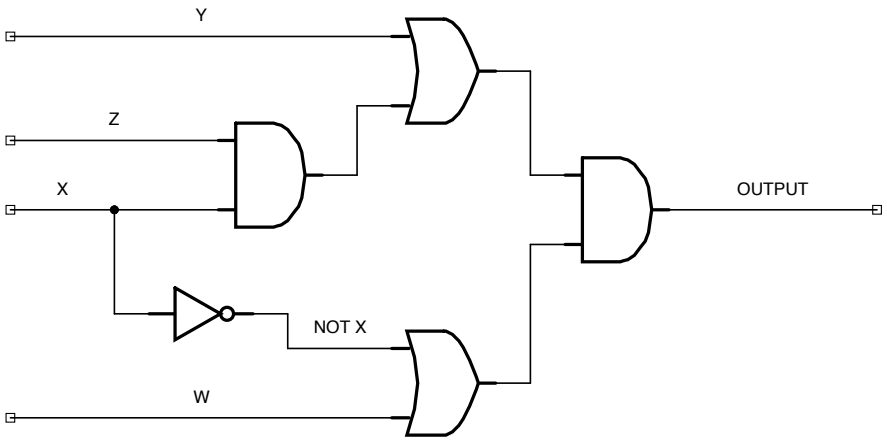
A	B	$!A$	$!B$	A && B	$!(A\&\&B)$	$(!A||!B)$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

   (b) Law 2: Stated in english and in 'C'

   English: Not A or B is the same as Not A and Not B
   'C': $!(A||B) == (!A\&\&!B)$

A	B	$!A$	$!B$	$A\|\|B$	$!(A\|\|B)$	$(!A\&\&!B)$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

2. Only using NOT, OR and AND gates draw the logic gate cicuit for the following boolean expression. $(w||!x)\&\&(y||(x\&\&z))$



PART B: Data Representation

1) Book problems

1. Solve problem 2.61, on page 121, from our CS:APP2e text.

Problem 2.61 Solution:
These exercises require thinking about the logical operation `!` in a nontraditional way. Normally we think of it as logical negation. More generally, it detects whether there is any nonzero bit in a word. In addition, it gives practice with masking.

```
A.!!x
B.!!~x
C.!!(x & 0xFF)
D.!!(~x & (0xFF << ((sizeof(int)-1)<<3)))
```

2. Solve problem 2.68, on page 123, from our CS:APP2e text.

Problem 2.68 Solution:

Here is the code:

code/data/bits.c

```
1 /*
2  * Mask with least signficant n bits set to 1
3  * Examples: n = 6 --> 0x2F, n = 17 --> 0x1FFFF
4  * Assume 1 <= n <= w
5  */
6 int lower_one_mask(int n) {
7     /*
8      * 2^n-1 has bit pattern 0...01..1 (n 1's)
9      * But, we must avoid a shift by 32
10     */
11    return (2<<(n-1)) - 1;
12 }
```

code/data/bits.c

The code makes use of the trick that `(1<<n)-1` creates a mask of `n` ones. The only challenge is to avoid shifting by w when `n` $= w$. Instead of writing `1<<n`, we write `2<<(n-1)`. This code will not work for `n` $= 0$, but that's not a very useful case, anyhow.

3. Solve problem 2.71, on page 124, from our CS:APP2e text.

Problem 2.71 Solution:

This problem highlights the difference between zero extension and sign extension.

A. The function does not perform any sign extension. For example, if we attempt to extract byte 0 from word `0xFF`, we will get 255, rather than -1.

B. The following code uses the trick shown in Problem 2.23 to isolate a particular range of bits and to perform sign extension at the same time. First, we perform a left shift so that the most significant bit of the desired byte is at bit position 31. Then we right shift by 24, moving the byte into the proper position and performing sign extension at the same time.

code/data/xbyte.c

```
1 int xbyte(packed_t word, int bytenum) {
2     int left = word << ((3-bytenum) << 3);
3     return left >> 24;
4 }
```

code/data/xbyte.c

4. Solve problem 2.76, on page 126, from our CS:APP2e text.

Problem 2.76 Solution:

Patterns of the kind shown here frequently appear in compiled code.

A. $K = 17$: `(x<<4) + x`
B. $K = -7$: `-(x<<3) + x`
C. $K = 60$: `(x<<6) - (x<<2)`
D. $K = -112$: `-(x<<7) + (x<<4)`

2) 2's Complement Respresentation

Fill in the below table assuming a 32 bit computer that uses 2's complement representation, `INT_MAX` and `INT_MIN` are defined as the computer's signed integer representation maximum and minimum value respectively, and:

```
int x = -1, y = 0xfeedface, z = INT_MAX, i = sizeof(short *);
```

C Expression	Hexadecimal
`x`	`0xffffffff`
`y`	`0xfeedface`
`z`	`0x7fffffff`
`i`	`0x00000004`
`z<<3`	`0xfffffff8`
`z<<((i>>1)-1)`	`0xfffffffe`
`~0 == (z + INT_MIN)`	`0x00000001`
`y & 0xffff`	`0x0000face`
`y >> 16`	`0xfffffeed`
`(y >> 16) | 0xffff`	`0xffffffff`
`(~(0x10>>2)+1) == (x*i)`	`0x00000001`
`(~z+1) + -1`	`0x80000000`
`(~((~x) << 1)) & y`	`0xfeedface`
`((y<<3)+INT_MIN)^((y<<3)+INT_MIN)`	`0x00000000`

The following code can be used to verify the answers:

```
#include <stdio.h>
#include <limits.h>

int
main(int argc, char **argv)
{
  int x = -1, y = 0xfeedface, z = INT_MAX, i = sizeof(int);

  printf("0x%08x\n", x);
  printf("0x%08x\n", y);
  printf("0x%08x\n", z);
  printf("0x%08x\n", i);
  printf("0x%08x\n", z<<3);
  printf("0x%08x\n", z<<((i>>1)-1 ));
  printf("0x%08x\n", ~0 == (z + INT_MIN));
  printf("0x%08x\n", y & 0xffff);
  printf("0x%08x\n", y >> 16);
  printf("0x%08x\n", (y>>16) | 0xffff);
  printf("0x%08x\n", (~(0x10>>2)+1) == (x*i));
  printf("0x%08x\n", (~z+1) + -1);
  printf("0x%08x\n", (~((~x)<<1)) & y);
  printf("0x%08x\n", ((y<<3)+INT_MIN) ^ ((y<<3)+INT_MIN));

  return 0;
}
```

3) Misc

1. $(010101)_2$ to base 10

   `21`

2. $(011100)_2$ to base 16

   `0x1C`

3. $(54.125)_{10}$ to binary

   `110110.001`

4. $(122.3)_8$ to base 16

   `0x52.4`

5. Find the decimal equivalent of the five-bit twos complement number: 11111

   `-1`

6. Show the results of adding the following pairs of six-bit twos complement numbers in decimal and indicate whether or not overflow occurs for each case.

   (a) 111110 + 111101

   `(-2 + -3) : bitwise sum 1111011 truncates to 111011 = -5 NO OVERFLOW`

   (b) 110111 + 110111

   `(-9 + -9) : bitwise sum 1101110 truncates to 101110 = -18 NO OVERFLOW`

   (c) 111111 + 001011

   `(-1 + 11) : bitwise sum 1001010 truncates to 001010 = 10 NO OVERFLOW`

7. Complete the following table for the 5-bit 2's complement representation. Show your answers as signed base 10 decimal integers and the 2's complement binary value.

value	decimal	binary
Largest Positive Number	`15`	`1111`
Most Negative Number	`-16`	`10000`
Number of distinct Numbers	32	