Princeton University
COS 217: Introduction to Programming Systems
Spring 2015 Midterm Exam Answers

The exam was a 50-minute, closed-book, closed-notes exam.

Question 1: Expressions in C

(a) True. The bit-wise complement of 1 is non-zero, 1 is non-zero, so their logical AND is true.

(b) False. 01000 is octal representation for the decimal number 512, so their difference is zero. Note that you cannot interpret 01000 as the decimal number 1000, due to the leading '0' which signifies an octal number.

(c) True. 0x2B is non-zero, and its bit-wise OR with anything is non-zero, even though the logical NOT of 0x2B (non-zero) is zero.

(d) True. 16 is 10000 in binary, so 16 >> 4 is 1.

(e) False. sizeof(5) is same as sizeof(int), sizeof(2L) is same as sizeof(long). The C language specification says that size of an int must be less than or equal to size of a long. This must hold on all systems, not only on nobel.

(f) False. The less-than operator has left-to-right associativity. Therefore, -10 < i < -1 is the same as (-10 < i) < -1. (-10 < i) evaluates to either 0 or 1, depending on the value of i. Since both 0 and 1 are greater than -1, the complete expression is False in either case.

Question 2: Matching Terms

(k)	Pass by value	
(i)	Pass by reference	
(j)	Modulo arithmetic	
(h)	Dangling pointer	
(a)	Memory leak	
(n)	External testing	
(l)	Regression testing	
(g)	Debugging heuristic	Alternate: (b) Debugging heuristic
(m)	Modularity heuristic	
(e)	Abstract data type	

Question 3: Short Answer

(a)
```
Binary representation of 30 as an 8-bit word:  00011110
Ones' complement of above:                      11100001
Two's complement = Ones' complement + 1:        11100010
```

Therefore, -30 is represented as 11100010 on a system with two's complement representation and an 8-bit word size.

Alternate way to get the same solution: To derive the two's complement of a given binary number, start scanning leftward from the least significant bit (the rightmost bit). Keep the 0's as they are, until you see the first 1. Keep the first 1 as is. After that, complement every bit you see.

(b)

```
Binary representation of 12 (8-bits is enough):       00001100
Binary representation of 34 (8-bits is enough):       00100010
(12 ^ 34) = bit-wise xor of above two numbers:        00101110
Above ^ 34:                                           00001100
Therefore, answer is 00001100, i.e., 12 in decimal.
```

Alternate way to get the same solution:
The bit-wise xor (^) operator is associative,
i.e., (12^34)^34 = 12^(34^34).
Any number xor itself is 0, i.e., x^x = 0.
Any number xor 0 is the number itself, i.e., x^0 = x.
So, a shortcut to the answer is (12^34)^34 = 12^(34^34) = 12^0 = 12.

(c) *p is assigned decimal number 10.
Note that malloc allocates only 1 byte, i.e., p points to 1 byte in memory, which is fine since p is declared as a pointer to a char.
When *p is assigned to a hexadecimal literal that is too long to fit in 1 byte, *p takes the truncated value from the lower byte, i.e., *p is assigned 0x0A, which is 10 in decimal.

(d) Expected answer:

```
unsigned int hash(const char *s)
{
   int i;
   unsigned int h = 0U;

   for (i=0; s[i] != '\0'; i++)
       h = h * 65599U + (unsigned int) s[i];

   return h;
}
```

Note that strlen(s) requires at least *n* operations for a string with length *n*. Calling it in the loop condition means you are doing *n* operations each time. The loop is executed *n* times, resulting in about n^2 operations in all. In contrast, checking the loop condition with the terminating character in the string is a constant-time operation, resulting in about *n* operations in all. This improves performance.

A partial solution is to recognize that strlen(s) does not change inside the loop, and it can be called one time before starting the loop. Still, this is not as good as the above solution, because you are still doing an extra call with *n* operations and there is some overhead in making a function call.

(e) The set of words accepted by the DFA are binary words that start with a 1 AND end with a 0. (Aside: examples of accepted words are 10, 100, 110, ... ; examples of rejected words are (empty word), 0, 1, 00, 01, 11, ...)

(f) The set of numbers accepted by the DFA are negative even integers. (Range between minimum and maximum is not needed, but minimum is -128 and the maximum is -2 for 8-bit words.)

Question 4: Bug Hunt

```
Bug 1 line number:  4
Fixed statement:    enum {MAX_WORD_LENGTH = 50};
Missing semi-colon.

Bug 2 line number:  9
Fixed statement:    char str1[MAX_WORD_LENGTH+1], str2[MAX_WORD_LENGTH+1];
We need an extra byte to store the terminating character in a string.

       Alternate:   4
       Fixed statement: enum {MAX_WORD_LENGTH = 51};
       This will achieve the same effect of correctly handling 50-letter words.

Bug 3 line number:  11
Fixed statement:    if (scanf("%s %s", str1, str2)!= 2)
Note that the compiler will not complain about the original statement, since %c
matches with the expected type (char *) of str1 and str2. However, the words
will not be read correctly into str1 and str2.

Bug 4 line number:  36
Fixed statement:    freq2[s2[i] - 'a']++;
This is a typical example of a copy-and-paste error.

Bug 5 line number:  39
Fixed statement:    for (i = 0; i < 26; i++)
This loop checks whether the frequency of each letter is the same in the two
words. We need to loop over all letters in the alphabet, not over all letters
in the words.

Bug 6 line number:  40
Fixed statement:    if (freq1[i] != freq2[i])
The result is 0, i.e., the words are NOT anagrams, if the frequency of any
letter is different in the two words.
```

Question 5: Abstract Data Type

(a) The type struct Bag is declared in bag.c for encapsulating data, i.e., clients cannot directly access the fields of a Bag object.

Its benefits (noted in lecture #10) are:
1. Clarity: encourages abstraction
2. Security: clients cannot corrupt object by changing its data in unintended ways
3. Flexibility: allows implementation to change – even the data structure – without affecting clients.

(b) Using void * makes the Bag object generic, i.e., it can contain (address of) any type of item, not just (address of) an int item. As mentioned in the description of the Bag ADT, there are no restrictions on the types of items a bag may contain.

Using const provides security to the clients of Bag ADT, that the value of an item in the bag cannot be changed by any operation in the Bag implementation. In other words, if any function in the Bag implementation tries to change the value of the item, there will be a compiler error.

(c) Expected answer:

```
int Bag_empty(Bag_T oBag) {
   assert(oBag != NULL);
   return oBag->psFirst == NULL;
}
```

(d) Yes, it is possible to tell from the given function declaration that the Bag_add_item operation does not check for duplicate items. This is because it does not accept a pointer to a comparison function. Without such a comparison function, it would not know how to compare its parameter to the item contained in the Bag object.

(e) Expected answer:

```
int Bag_count_item(Bag_T oBag, const void *pvItem,
   int (*pfCompare)(const void *pvItem1, const void *pvItem2)) {
   struct BagNode *psNode;
   int iCount = 0;

   assert(oBag != NULL);
   assert(pfCompare != NULL);
   assert(pvItem != NULL); /* optional */

   for (psNode = oBag->psFirst; psNode != NULL; psNode = psNode->psNext)
      if ((*pfCompare)(psNode->pvItem, pvItem) == 0)
         iCount++;
   return iCount;
}
```