15-122 : Principles of Imperative Computation, Spring 2013

Homework 7 Theory [Update 1]

Due: Tuesday, April 16, 2013, at the **beginning** of lecture

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework will give you some practice working with issues in C code, integer types, and tries. You can either type up your solutions or write them *neatly* by hand in the spaces provided. You should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

Update 1, April 9: Fixed a typo in question 2.

Question	Points	Score
1	9	
2	10	
3	6	
Total:	25	

You *must* use this printout, include this cover sheet, and staple the whole thing together before turning it in. Either type up the assignment using 15122-theory7.tex, or print this PDF and write your answers *neatly* by hand.

1. **C programming issues**

   For each of the following problems:

   1. **state what is wrong with the code** and;
   2. **show how to correct it**.

   Do not just try to compile it and write down the error message. (Compiling and running the code may be useful, but all of these examples will compile without error, and some will even run and produce output, but they all contain conceptual errors that affect correctness, causing either an exception or undefined behavior at runtime.) Read the code and explain what is being done wrong, conceptually.

   Think about all the ways to incur undefined behavior in C, including accessing unallocated or uninitialized memory, dereferencing `NULL`, dividing by zero, and signed arithmetic overflow.

(1) (a) The `strlen` function returns the length of a standard C string. For example, the call `strlen("foo")` will return 3. If the `char` array passed to `strlen` is not a valid, `'\0'`-terminated C string, `strlen` will cause undefined behavior.

The `strcpy(arr,str)` function copies a string of length n (the second argument `str`) into a `char` array that has length *strictly greater* than n (the first argument `arr`). If the array is not big enough, `strcpy` will cause undefined behavior.

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

int main() {
  char w[strlen("C programming")];
  strcpy(w,"C programming");
  printf("%s\n", w);
  return 0;
}
```

Solution:

(1) (b)
```
#include "xalloc.h"

int main() {
  int* a = xmalloc(100*sizeof(int));
  for (int i=0; i<100; i++) {
    if (a[i] == 0) a[i]=i;
    else a[i]=0;
  }
  free(a);
  return 0;
}
```

Solution:

(1) (c)
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char name[strlen("wordpress")];
  strcpy(name,"wordpress"+1);
  printf("%s\n", name);
  free(name);
  return 0;
}
```

Solution:

(1) (d)
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xalloc.h"

int main() {
  char *letter_data = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char *a = xcalloc(16, sizeof(char));
  for (int i = 0; i < 16; i++) {
    a[i] = letter_data[i];
  }
  printf("The first sixteen letters are: %s\n", a);
  free(a);
  return 0;
}
```

Solution:

(1) (e) This code fragment shows a C function that is called from another function. You can assume the precondition will be satisfied.

```
#include "contracts.h"
#include "xalloc.h"

// Returns an array of length n that must be freed by the caller
int *square_ar(int n)
{
  REQUIRES(n >= 0);
  int *A = xcalloc(n, sizeof(int));
  for (int i = 0; i < n; i++)
    A[i] = i * i;
  return A;
}
```

Solution:

(1) (f) This code fragment shows a C function that is called from another function.

```
#include <stdio.h>
#include <string.h>
#include "contracts.h"
#include "xalloc.h"

char *rev_str(char *str) {
  char *rev = xcalloc(strlen(str)+1, sizeof(char));
  for (int i = strlen(str); i >= 0; i--) {
    rev[strlen(str) - i] = str[i];
  }
  printf("The reversed string is: %s\n", rev);

  ENSURES(strlen(rev) == strlen(str));
  return rev;
}
```

Solution:

(1) (g) This code fragment constructs a string of length `n` with the character `c` repeated that many times. The function given here will likely be called from other functions many times over.

```
char *strcdup(int n, char c) {
  char dup[n+1];
  int i;
  for (i = 0; i < n; i++)
    dup[i] = c;
  dup[i] = '\0';
  char *A = dup;
  return A;
}
```

Solution:

(1) (h) This code fragment shows a C function that is used inside the library implementation of queues. Assume that `is_queue` returns true.

```
#include <stdlib.h>
#include "contracts.h"
#include "xalloc.h"

int queue_size(queue Q) {
  REQUIRES(is_queue(Q));
  list *L = xmalloc(sizeof(struct list_node));
  int size = 0;
  for (L = Q->front; L != Q->back; L = L->next) {
    size++;
  }
  free(L);
  return size;
}
```

Solution:

(1) (i) This code fragment is intended to free a struct and also print out the contents of the struct if the second argument `also_print` is `true`. You can assume that the pointer passed to the function points to a heap-allocated memory block and that the strings do not need to be freed.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "xalloc.h"

struct record_header {
  int num;
  char* name;
};
typedef struct record_header record;

void free_also_print(record *p, bool also_print)
{
  REQUIRES(p != NULL);
  if (also_print) {
    printf("Number: %d \n", p->num);
    printf("Name: %s \n", p->name);
    free(p);
  }
  free(p);
  return;
}
```

Solution:

2. **Casting**

In C, we frequently have to cast types on the basis on their sign and size. Doing so can change the bit-patterns and their interpreted values. This question will test your understanding of how casting works.

Suppose that we are working with the expected implementation-defined implementation of unsigned and signed (2's compliment) `char` (8 bits, one bite), `short` (16 bits, two bytes), and `int` (32 bits, four bytes). Also assume that `char` is a `signed char`, though this is implementation-defined.

(7) (a) We begin with the following declarations:

```
short x = -6;
unsigned short y = 246;
char z = -2;
unsigned char w = 250;
```

Fill in the table below. In the third column, always use two hex digits to represent a `char` (signed or unsigned), four hex digits to represent a `short`, and eight hex digits to represent an `int`.

C expression	**Decimal value**	**Hexadecimal**
`x`	-6	`0xFFFA`
`(unsigned short)x`	65530	`0xFFFA`
`(int)x`	-6	`0xFFFFFFFA`
`y`	246	`0x00F6`
`(short)y`		
`(unsigned int)y`		
`z`	-2	
`(unsigned char)z`		
`(int)z`		
`w`	250	
`(char)w`		
`(unsigned short)w`		

(3) (b) Say we have a (signed) `char` array of length 4 and we want to store that array in a single (4-byte) `int` by storing the `char` array `{1, 2, 3, 4}`, for example, as `0x01020304`.

Write a C function that takes a length-4 `char` array named `A` and squeezes it into a single `int` as outlined above. Do not cast directly between signed and unsigned types of different sizes, and make sure your solution works for `char` arrays containing negative values.

Your solution should be clear and straightforward; convoluted code will lose points.

Solution:

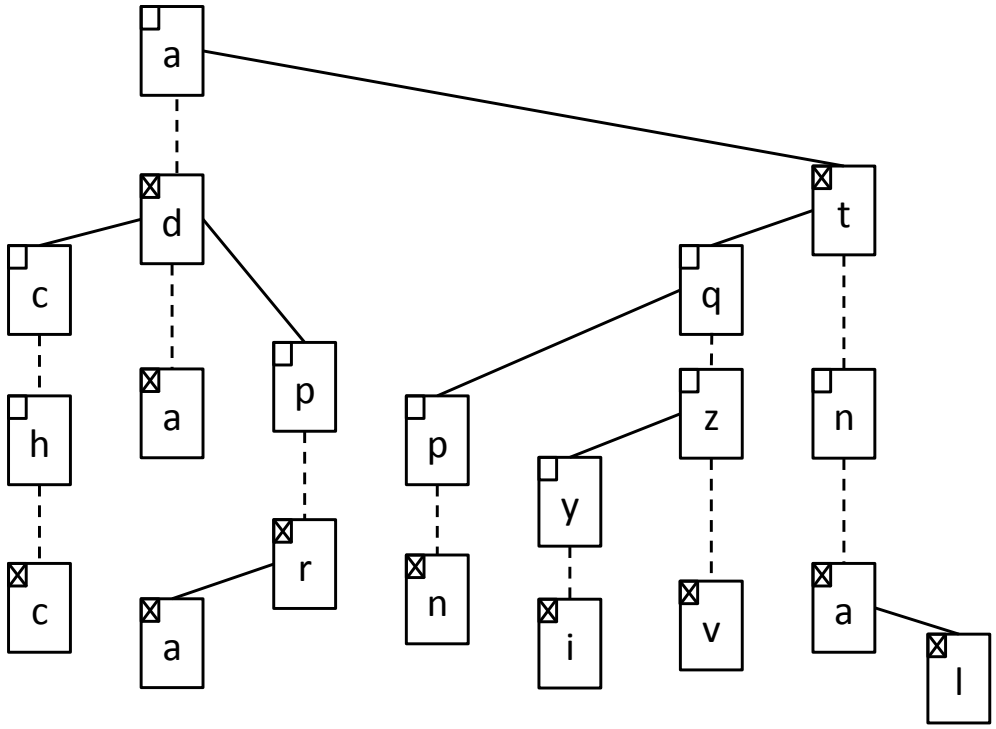
```
int squeeze(char *A) {
  // requires that A have length >= 4




}
```

3. **Tries and tries again**

DO NOT BE CONFUSED by the deceptive appearance of a festive holiday message! As a ternary search trie, this data structure contains mostly nonsense words:



As in lecture, the dotted lines connect a node to its `middle` child, and solid lines connect a node to its `left` and `right` children.

(1) (a) What are the *first four* (4) strings contained in this TST, alphabetically?

Solution:

(1) (b) What are the *last four* (4) strings contained in this TST, alphabetically?

Solution:

(1) (c) How many strings are there total?

Solution:

(3) (d) The ternary search trie below contains real words. Add the words `adage`, `on`, `sage`, `ski`, `slow`, and `so` to this data structure.

Solution: