# 15-122 : Principles of Imperative Computation

## Fall 2012

## Assignment 7, Update 1

(Theory Part)

Due: Tuesday, November 20, 2012 at the **beginning** of lecture

Name: ______________________________

Andrew ID: ______________________________

Recitation: ______________________________

The written portion of this week's homework will give you some practice working with hash tables, heaps and priority queues. 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, November 14 – fixed typo in 2(a)

| Question | Points | Score |
|---|---|---|
| 1 | 8 | |
| 2 | 5 | |
| 3 | 7 | |
| 4 | 5 | |
| Total: | 25 | |

You *must* include this cover sheet.
Either type up the assignment using 15-122-theory7.tex, or print out this PDF.

1. **C programming issues**

   For each of the following problems, state what is wrong with the code and show how to correct it. Do not just try to compile it and write down the error message. (Some of these will compile without error, and some will even run and produce output, but they all contain conceptual errors that may affect correctness.) 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 arithmetic overflow.

(1) (a)

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

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

**Solution:**

(1) (b)

```
#include "xalloc.h"

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

**Solution:**

(1) (c)
```
#include "xalloc.h"
#include <string.h>

int main() {
  char* name = xmalloc(strlen("wordpress"+1));
  strcpy(name,"wordpress");
  return 0;
}
```

**Solution:**

(1) (d) The standard string library function `strncpy(dest, src, n)` copies the specified number of characters `n` from the source string `src` to the destination string `dest`.

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

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

**Solution:**

(1) (e) This code fragment shows a C function that is called from another function. It is supposed to return the result only if no overflow occurs.

```
#include <assert.h>

int oadd(int x, int y) {
   int result = x + y;
   if (x > 0 && y > 0) assert(result > 0);
   if (x < 0 && y < 0) assert(result < 0);
   return result;
}
```

**Solution:**

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

```
#define TABLESIZE 100
int table[TABLESIZE];

int insert_in_table(int pos, int value) {
  if (pos >= TABLESIZE)
    return -1;
  table[pos] = value;
  return 0;
}
```

**Solution:**

(1) (g) This code fragment shows a C function that is used inside the library implementation of stacks. Assume that `is_stack` returns true.

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

int stack_size(stack S) {
  REQUIRES(is_stack(S));
  list *L = xmalloc(sizeof(struct list_node));
  int size = 0;
  for (L = S->top; L != S->bottom; L = L->next)
    size++;
  return size;
}
```

**Solution:**

(1) (h) This code fragment shows a C function that applies a function to an array of length `n`. We would expect it to be called from another function.

```
#include <stdlib.h>

int* apply_fn(int (*f)(int* x, int y), int n, int A[n]) {
  int accum = 0;
  int* result = NULL;
  for (int i = 0; i < n; i++) {
    accum = (*f)(&accum, A[i]);
  }
  result = &accum;
  return result;
}
```

**Solution:**

2. **Pass by reference**

   At various points in our C0 programming experience we had to use somewhat awkward workarounds to deal with *functions that need to return more than one value*. The address-of operator in C gives us a new way of dealing with this issue.

(2) (a) Sometimes, a function needs to be able to both 1) signal whether it can return a result and 2) return that result, if it was able. One such function was `peg_solve`, which either returned a series of moves on the initially-empty stack we passed in or else returned `false`. Parsing also fits this pattern. Consider the following code:

```
bool my_int_parser(char *s, int *i);

void parseit(char *s) {
  REQUIRES(s != NULL);
  int *i = xmalloc(sizeof(int));
  if(my_int_parser(s, i))
    printf("Success: %d.\n", *i);
  else
    printf("Failure.\n");
  free(i);
  return;
}
```

Using the address-of operator, rewrite the body of the `parseit` function so that it does not heap-allocate, free, or leak any memory on the heap.

**Solution:**

```
bool my_int_parser(char *s, int *i);

void parseit(char *s) {
  REQUIRES(s != NULL);


  return;
}
```

(3) (b) It is possible to return multiple values by C and C0 by bundling them in a struct:

```
struct bundle { int x; int y; };

struct bundle *foo(int x) {
  ...
  struct bundle *B = xmalloc(sizeof(struct bundle));
  B->x = e1;
  B->y = e2;
  return B;
}

int main() {
  ...
  struct bundle *B = foo(e);
  int x = B->x;
  int y = B->y;
  free(B);
  ...
}
```

Rewrite the declaration and the last few lines of the function `foo`, as well as the snippet of `main`, to avoid heap-allocating, freeing, or leaking any memory on the heap. The rest of the code should continue to behave exactly as it did before.

**Solution:**

```
__________ foo(______________________________________________) {
  ...

  ___________________________________________________________

  ___________________________________________________________

  ___________________________________________________________
}
int main() {
  ...

  ___________________________________________________________

  ___________________________________________________________

  ___________________________________________________________
  ...
}
```

3. **Strings: be careful about complexity!**

(1) (a) The string buffer implemented in the theory homework is essentially an unbounded array. In a sentence or two, give one good reason why we might want string buffers instead of a *generic* implementation of unbounded arrays (like the generic implementations of stacks or priority queues we have considered).

**Solution:**

(3) (b) In C, the function `strlen(s)` has a runtime in $O(n)$, where $n$ is the length of `s`.

```
for (int i = 0; i < strlen(s); i++)
  if (s[i] == 'A') return true;
```

What's the worst-case runtime (big-O) of this loop?

**Solution:**

Rewrite this code so that it will do the same thing faster:

**Solution:**

What is the worst-case runtime (big-O) of your revised code?

**Solution:**

(3) (c) In C0, the function `string_join(x,y)` has a runtime in $O(m)$, where $m$ is the sum of the sizes of `x` and `y`, because both strings need to be copied into new memory. In general, the optimal runtime for joining any number of strings with *total* length $m$ is in $O(m)$ – so the same upper bound should apply whether we're joining ten thousand strings of size 10 or 10 strings of size ten thousand.

Your task is to implement a function `char *string_join_all(char **S, int n)` that takes an array of `n` strings and returns a newly-allocated string that concatenates all of the `n` strings in the array, taking time in $O(m)$. String buffers would help us do this, but your solution should *not use string buffers* or reimplement the infrastructure of string buffers.

For comparison, here is a function that returns the correct result, but does *not* have the correct worst-case runtime when asked to concatenate many small strings.

```
char *string_join_all(char **S, int n)
{
  REQUIRES(n >= 0);

  char *result = xcalloc(1, sizeof(char));
  result[0] = '\0';

  for (int i = 0 ; i < n ; i++)
  {
    char *si = S[i];
    int result_length = strlen(result);
    int li = strlen(si);

    char *new_result = xcalloc(result_length + li + 1, sizeof(char));

    strcpy(new_result, result);
    strcpy(new_result + result_length, si);

    free(result);
    result = new_result;
  }

  return result;
}
```

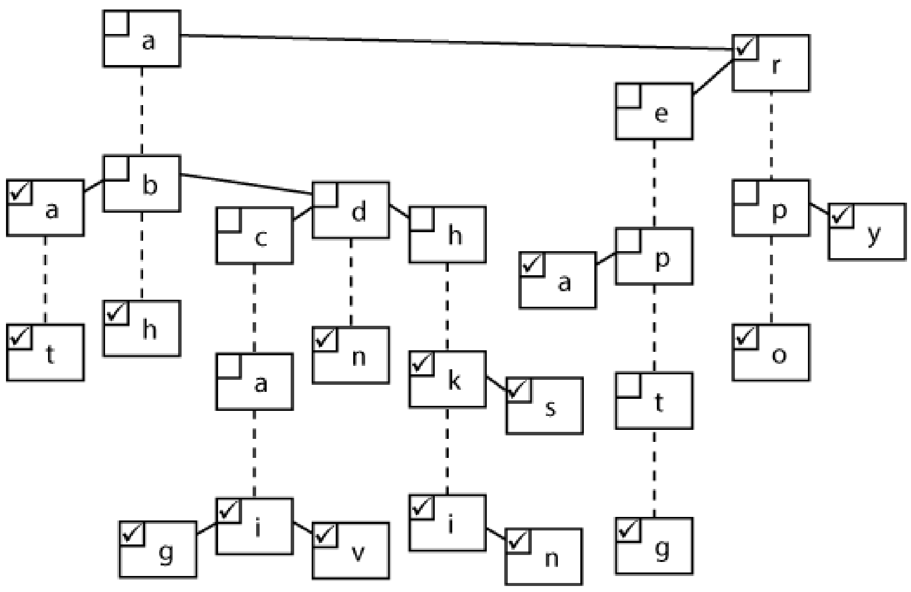
A few hints. First, looping through the array of strings more than once might be worth it if it saves you time later on. Second, it is likely that we will not give full credit to solutions that allocate repeatedly.

**Solution:**

```
char *string_join_all(char **S, int n)
{
  REQUIRES(n >= 0);
















}
```

4. **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?

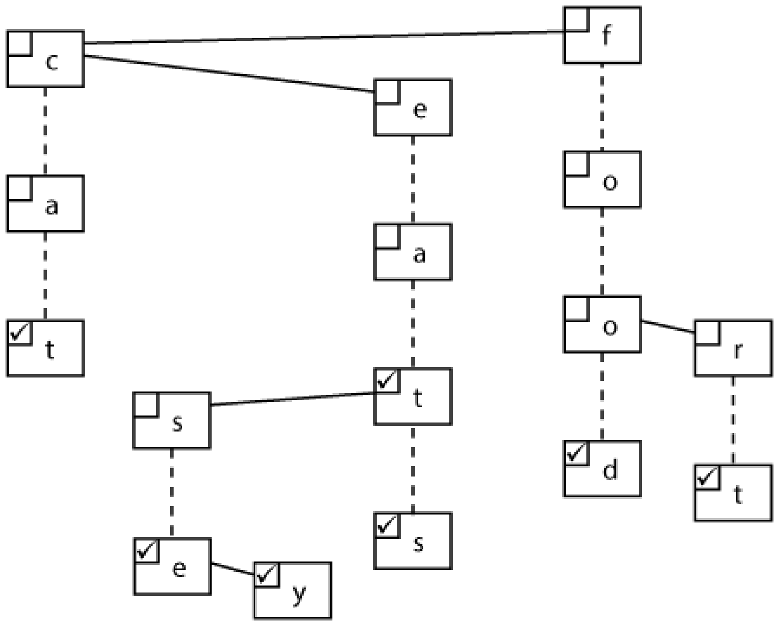
**Solution:**

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

**Solution:**

(2) (d) The ternary search trie below contains real words. Add the words `cake`, `east`, `for`, and `soup` to this data structure.

**Solution:**


c
a
t
e
a
t
s
s
e
y
f
o
o
r
d
t