This exam is based on CS107 exams from previous years. It does not include a memory drawing problem, so see Practice #1 or Practice #3 for one of those, and expect one on the exam! --Cynthia

CS107
Spring 2017

CS107 Midterm #1 Examination (Practice #2)

This is a closed book, closed note, closed computer exam. You have 120 minutes to complete all problems. You don’t need to \#include any libraries, and you needn’t use assert to guard against any errors. Understand that the majority of points are awarded for concepts taught in CS107, and not prior classes. You don’t get many points for for-loop syntax, but you certainly get points for proper use of &, *, and the low-level C functions introduced in the course.

Good luck!

Name: ___________________

SUNet ID: ___________________
Problem 1: **strseparate**

You will implement the following function:

```c
char *strseparate(char **stringp, const char *delimiters);
```

Provided **strseparate** is properly implemented, this program

```c
static void printTokens(const char *sentence) {
    char copy[strlen(sentence) + 1];
    strcpy(copy, sentence);
    char *curr = copy;
    while (curr != NULL) {
        char *word = strseparate(&curr, ". !"); // note curr is updated by reference
        printf("%s\n", word);
    }
}

int main(int argc, const char *argv[]) {
    printTokens("This is a sentence!!");
    return 0;
}
```

outputs the following:

```
"This"
"is"
"a"
"sentence"
"
```

**strseparate** examines the C string reachable from **stringp**, locates the first occurrence of any character (including the ' \0 ') in **delimiters**, and overwrites it with ' \0 ', thereby truncating the string. The address of the character following the freshly written ' \0 ' (or **NULL**, if the end of the C string addressed by **stringp** was reached) is stored in the space addressed by **stringp**, and the original **char** * addressed by **stringp** is returned.

Over the lifetime of the sample program above, the **copy** buffer evolves from this:

```
T h i s i s a s e n t e n c e ! ! \0
```

into this:

```
T h i s \0 i s \0 a \0 s e n t e n c e \0 \0 \0
```

and **curr**, which initially addresses the ' T ', will be updated to address the second ' i ', then the ' a ', and so on.
Problem 1 [continued]

Present your implementation of \texttt{strseparate} on the space below. The only \texttt{string.h} function you may (and, in fact, are required to) use is \texttt{strchr}, which is much like \texttt{strstr}, except that it searches for a character instead of a string.

\begin{verbatim}
char *strchr(const char *str, int c);
    The \texttt{strchr} function locates the first occurrence of \texttt{c} (converted to a \texttt{char}) in \texttt{str}, and returns a pointer to the located character (or \texttt{NULL} if the character isn’t present).
\end{verbatim}

Feel free to tear out the previous page so you can more easily refer to it.

\begin{verbatim}
static char *strseparate(char **stringp, const char *delimiters) {
\end{verbatim}
Problem 2: C-strings and client of generic interfaces (25 points)
We define a word's signature to be the string containing the letters of the word in sorted order. For example, the signature of "apple" is "aelpp" and the signature of "bell" is "bell". Write the make_map function below to take a CVector of words and fill a CMap with entries mapping each word (key) to its signature (value). The CVector was created using cvec_create(sizeof(char*), capacity, cleanup). You can assume the cleanup function is already correctly implemented. To form a word's signature, sort its letters into alphabetical order. You must use qsort and this will require that you write an appropriate callback function. You may assume the words consist only of lowercase letters.

CMap *make_map(const CVector *cv)
{
    CMap *cm = cmap_create(sizeof(char*), cvec_count(cv), cleanup);
Problem 3: The accumulate generic

Consider the following two functions, noticing that they have very similar structure:

```c
int int_array_product(const int array[], size_t n) {
    int result = 1;
    for (size_t i = 0; i < n; i++) {
        result = result * array[i];
    }
    return result;
}

double double_array_sum(const double array[], size_t n) {
    double result = 0.0;
    for (size_t i = 0; i < n; i++) {
        result = result + array[i];
    }
    return result;
}
```

You are to implement a generic accumulate function that captures the shared structure of these two functions. It takes the base address of an array and its effective length, the array element size, the function callback that should be repeatedly applied (above it would be multiply and add, but implemented as 2-argument functions rather than operators directly), the address of the default/starting value (to play the role of 1 and 0.0 in the above code), and the address where the overall result should be placed. The function type of the callback is as follows:

```c
typedef void (*BinaryFunc)(void *partial, const void *next, void *result);
```

Any function that can interpret the data at the first two addresses, combine them, and place the result at the address identified via the third address falls into the BinaryFunc function class. (The const appears with the second address, because it’s expected that the array elements—the elements that can’t be modified—be passed by address through the next parameter.)

a) First, implement the generic accumulate routine. We’ve provided some of the structure that should contribute to your implementation. You should fill in the three arguments needed so that memcpy can set the space addressed by result to be identical to the space addressed by init, and then go on to fill in the body of the for loop.

```
This first part was designed to expose basic memory and pointer errors very early on—e.g. to confirm that weren’t dropping &’s and *’s where they weren’t needed.
```

```c
void accumulate(const void *base, size_t n, size_t elem_size,
    BinaryFunc fn, const void *init, void *result) {
```
b) Now reimplement the `int_array_product` function from the previous page to leverage the `accumulate` function you just implemented for part a). Assume the name of the callback function passed to `accumulate` (which you must implement) is called `multiply_two_numbers`.

```c
static void multiply_two_numbers(void *partial, const void *next, void *result) {

int int_array_product(const int array[], size_t n) {
```
Problem 4: Integer Types

a) Write the unsigned binary number 101100111010 in hexadecimal: _____________________

b) Write the signed (two’s complement) binary number 101101 in decimal: ______________

c) Write the signed (two’s complement) binary number 001101 in decimal: ______________

d) Write the hexadecimal number 0x54BEEF as an unsigned binary number:

______________________________________________________

e) Write the decimal number 28 as an 8-bit binary number: ____________________________

f) Write the decimal number 28 in hexadecimal: _______________________________________

g) Write the decimal number -33 as an 8-bit signed (two’s complement) binary number:

________________________________________________________
CVector Functions

typedef int (*CompareFn)(const void *addr1, const void *addr2);
typedef void (*CleanupElemFn)(void *addr);
CVector *cvec_create(int elemsz, int capacity_hint, CleanupElemFn fn);
void cvec_dispose(CVector *cv);
int cvec_count(const CVector *cv);
void *cvec_nth(const CVector *cv, int index);
void cvec_insert(CVector *cv, const void *addr, int index);
void cvec_append(CVector *cv, const void *addr);
void cvec_replace(CVector *cv, const void *addr, int index);
void cvec_remove(CVector *cv, int index);
int cvec_search(const CVector *cv, const void *key,
                CompareFn cmp, int start, bool sorted);
void cvec_sort(CVector *cv, CompareFn cmp);
void *cvec_first(CVector *cv);
void *cvec_next(CVector *cv, void *prev);

CMap Functions

typedef void (*CleanupValueFn)(void *addr);
CMap *cmap_create(size_t valuesz, size_t capacity_hint, CleanupValueFn fn);
void cmap_dispose(CMap *cm);
int cmap_count(const CMap *cm);
void cmap_put(CMap *cm, const char *key, const void *addr);
void *cmap_get(const CMap *cm, const char *key);
void cmap_remove(CMap *cm, const char *key);
const char *cmap_first(const CMap *cm);
const char *cmap_next(const CMap *cm, const char *prevkey);

Other Functions

void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
void *memset(void *ptr, int value, size_t num);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
size_t strlen(const char *s);
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
char *strdup(const char *s);
char *strcat(char *dest, const char *src);
char *strchr(char *str, int character);
char *strtok(char *str, const char *delimiters );
int sprintf ( char * str, const char * format, ... );
int strcmp(const char *str1, const char *str2);
int strncmp(const char *str1, const char *str2, size_t n);
int strncmp(const char *str1, const char *str2, size_t n);