CS107 Handout 24
Spring 2008 May 6, 2008

Section Handout

Problem 1: South of Market (CS107 Midterm: Last Autumn)

Consider the following **`struct`** definitions:

```
   typedef struct alley {
      char clementina[4];
      short **minna[3];
      struct alley *jessie;
      int clara;
   } alley;

   char *washburn(alley *grace, short *bernice)
   {
line 1    grace[2].clementina[12] = *bernice;
line 2    ((alley *)(grace->minna))->jessie[2].clara += 960;
line 3    return *(char **)washburn(grace + 2, &bernice[2]);
   }
```

Generate code for the entire **`washburn`** function. Be clear about what assembly code corresponds to what line.

Problem 2: Matchmaking (CS107 Final Exam: Spring 2006)

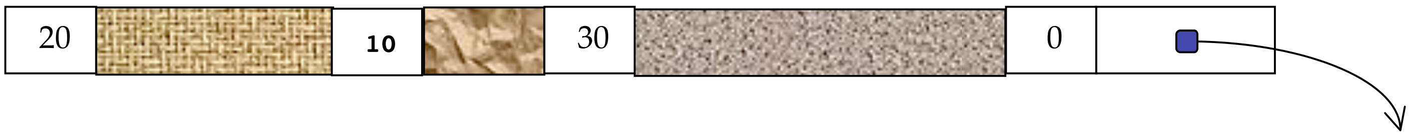
You keep the names of all your male friends in one C **`vector`** (where else?) and the name of all your female friends in a second C **`vector`**. Your task is to generate a brand new **`vector`** and populate it will the full cross product of men and women as **`char *`**, **`char *`** pairs.

Write a function **`generateAllCouples`** that creates a **`vector`**, inserts all boy-girl pairs, and then returns it.

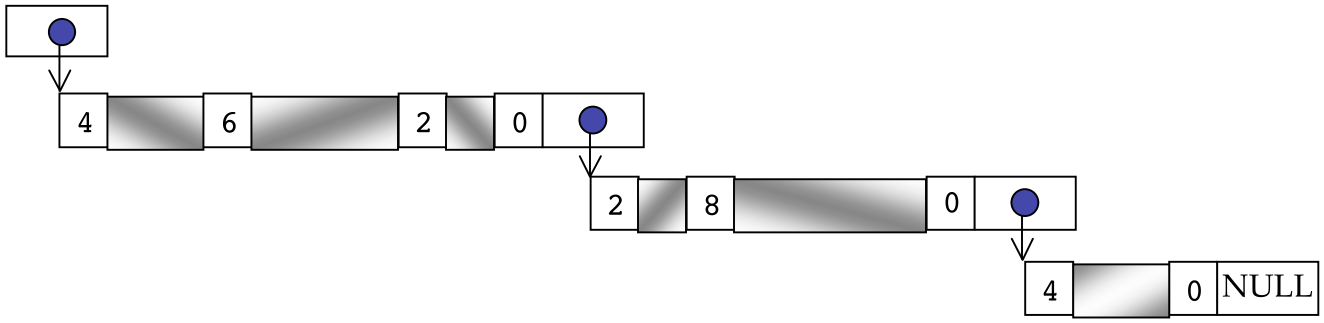
```
   /**
    * The primary assumption is that both boys and girls
    * are C vectors of dynamically allocated C strings, each
    * initialized as follows.
    *
    *   vector boys, girls;
    *   VectorNew(&boys, sizeof(char *), StringFree, 0);
    *   VectorNew(&girls, sizeof(char *), StringFree, 0);
    *
    * generateAllCouples creates a new C vector of couples
    * and inserts one such record on behalf of every possible
    * mapping of boy to girl.  The couples own their own strings,
    * so that none of the three vectors share any memory whatsoever.
    *
    * Assume that CoupleFree is the VectorFreeFunction that disposes
    * of couple records embedded in a vector, and assume it just works.
    */

   typedef struct {
      char *girl;
      char *boy;
   } couple;

   vector generateAllCouples(vector *boys, vector *girls)
   {
      vector couples;
      VectorNew(&couples, sizeof(couple), CoupleFree, 0);
```

Problem 3: `packPackets`

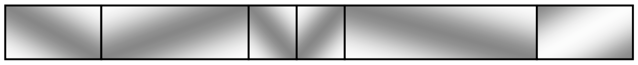
You're to write code that traverses a linked list of custom nodes and builds a dynamically allocated array of bytes large enough to store the meaningful data held by the linked list. Each node stores one or more packets of raw information, where each packet is preceded by a two-byte short stating the length (in bytes) of the packet that follows it. The end of the packet sequence is marked by a two-byte zero, and the four bytes after that store the address of the next node in the list (or **`NULL`**, if the node is the last in the list). The following node contains three packets of length 20, 10, and 30 bytes, in that order. The number of bytes making up the entire node is 72.



You're to write a function called **`packPackets`** that, given the address of the first node in such a list, builds a dynamically allocated byte array where all packets in the linked list are replicated verbatim, one after another, in the same order they appear in the list. So, given the following list:



your **`packPackets`** function would return the address of the first byte of this dynamically allocated figure:



Here are some constraints I'm imposing on your implementation.

- The address passed to **`packPackets`** is the address of the first short storing the number of bytes making up the first packet in the first node.

- Each node alternates between shorts and packets. A two-byte zero marks the end of the packet sequence in any given node. Every node ends with a four-byte pointer identifying the location of the next node in the list.
- You should implement this function iteratively, and your algorithm should accomplish everything in exactly one traversal. This requires you to call **`realloc`** for each packet you encounter during your one traversal. [Tip: When **`NULL`** is passed as the first argument to **`realloc`**, **`realloc`** just calls **`malloc`** on your behalf.] This artificial constraint is being imposed so that I can test your understanding of **`realloc`**.
- For simplicity, you may assume the list contains at least one node, and that each node contains at least one packet.
- You needn't worry about any alignment restrictions at all.
- You shouldn't free or modify the original list in any way.

```
/**
 * Function: packPackets
 * ---------------------
 * Builds a contiguous array version of the packet list structure according
 * to the explanations provided on the prior page.  The parameter passed
 * is of type short *, because the first meaningful piece of information
 * stored in the list is a two-byte short.  The return value is of type
 * void *, because the implementation has no type information about the
 * packet data.
 */

void *packPackets(short *list)
{
```