CS-3411 Program I : Pack and Unpack

Spring 2020
Due: February 5, AoE

In this project, we will develop a relatively simple procedure which implements a mechanism through which we can pass an arbitrary number of function arguments although the function uses a fixed number of arguments. Assume that the function, hereafter referred to as *f* remembers no state (i.e., no global variables, no static variables) and it accepts three arguments, an integer followed by two `void *` arguments and returns a `void *` value :

```
void * f (int code, void * mem, void * data);
```

This function must be placed in a file called `f.c` which can be compiled and linked together with another file called `main.c` which invokes it.
The first integer argument (code) always assumes one of the values indicated by the following C defines :

```
#define  F_first        1   /* This is the first call to the function.  */
#define  F_last         2   /* This is the last call to the function. Free the memory area. */
#define  F_data_int     3   /* Void * argument points to integer data. */
#define  F_data_char    4   /* Void * argument points to character string. */
#define  F_data_float   5   /* Void * argument points to a float data value. */
#define  F_print        6   /* Print the accumulated values. */
```

If the first argument is `F_first` then the function should allocate an initial chunk of memory whose size is given by the *value* of the *data* argument (not the actual data it points to). The value of data argument cannot be zero. In this case, the procedure should not allocate any space and return a zero value. Otherwise, the function returns the address of the allocated area.

For example the following statement should be equivalent to a malloc of 200 characters:

```
  char * p;
  p = (char *)f(F_first, 0, (void *)200);
```

After the initial call, it is the responsibility of the user to pass the pointer to the allocated area as the second argument, namely, *mem.* Any successive calls with values other than `F_first` should store the value pointed to by the data argument in the allocated area, keeping track of remaining space in the allocated area. A final function call with `F_print` should print the accumulated data values. The following example should print the text :

```
System Programming class has 79.7 registered students in a classroom of 90
```

```
    int     i_a;
    float   f_a;
    void  * m;
    int   * ip;
    float * fp;

    fp = & f_a;
    ip = & i_a;
    m = f (F_first, 0, 256);     /* Allocate 256 bytes of data area. */
    m = f (F_data_char, m,  (void *)"System programming class has ");

    f_a = 79.7;
    m = f (F_data_float,  m,  (void *)fp);
    m = f (F_data_char, m,  (void *)"registered ");
    m = f (F_data_char, m,  (void *)"students in a ");
    m = f (F_data_char, m,  (void *)"classroom of ");
    i_a = 90;
    m = f (F_data_int,  m,  (void *)ip);
    m = f (F_data_char, m,  (void *)"\n");

    m = f (F_print, m, 0);
```

Note that your function should work correctly with any number of calls between a `F_first` and `F_last` and a `F_last` should free all the allocated area. Until the area is freed, multiple calls with `F_print` should print the same data multiple times.

Pragmatics

Achieving this functionality requires the function f to know what type of data item is stored in the area, and the total count of them. You need to reserve the first two bytes of the allocated area and use it as a pointer which points at the end of the filled area. Hence after an initial call these bytes would typically contain binary 2, which is the first usable byte. Similarly, each data item is preceded by a byte which indicates its type. You may also assume that the user of the function calculates this overhead when requesting the memory allocation.

Error Checking and Additional Constraints

Your function should not permit a NULL argument as *mem* unless it is the first call. Your function should require a NULL argument for *mem* argument when it is the first call. The function should check for the validity of the code argument.

If the user executes a *free* using the pointer returned that function instead of calling the function with the code `F_last`, execution of this free function call *must* free all area allocated by the procedure.

Submission Requirements

Your submission must be written in C. It is imperative that you follow this specification to the letter for your organization within the byte array used as mem in the project. Recapping, the first thing in the byte array must be 2 bytes that act as a pointer to the beginning of the part of mem that is free. These 2 bytes are a pointer, but not in the canonical sense. They are an offset that when added to the address of mem yields the address of the beginning of the free area within mem. For example, if mem is completely empty the first 2 bytes should contain the value 2. I.e., mem + 2 is the location where new data to add can be stored. The plus 2 is because the first 2 bytes are reserved for the offset. You must update this offset with each insertion to mem.

Every data entry must be accompanied by a 1 byte identifier that specifies the data's type. For an int this would be F_data_int which has a decimal value of 3. Following this identifier should be the 4 bytes that makeup the integer to be stored.

When you add a data element to mem, you must use big endian byte order. Place the most significant byte at the lowest memory address.

A call with code F_first must allocate a contiguous area for mem. You are not permitted to link several non-contiguous regions together. If I call free() instead of using F_last, the entire mem should be freed. This is the case if mem is 1 region.
In the main.c that is available with the project files there is a line m = f(F_First, 0, 0). This is for the purpose of testing f when 0 is specified for the allocation size. You should remove this for subsequent testing. The main will be replaced with a different code when grading your submission. The main included is for you to test your function f while developing to code.

Use Canvas to submit a tar file named `prog1.tgz` that contains:

1. A copy of the source file f.c **with comments**.
2. A copy of the test case main.c **with comments**.
3. A makefile which invoked generates a binary called f.o, a binary called main.o and a binary called a.out.
4. Running a.out should execute the above example.