Problem 1: Linked List Library (list)

In this problem we will learn how to write code distributed over multiple files that is compiled together and linked into one application. Download the zipped folder `http://web.mit.edu/6.s096/www/hw/2/list.zip` and take a look at the existing code.

First notice the directory structure. The zipped folder contains an `include/` directory, a Makefile which is much larger than the one we've been using in the past, a GRADER_INFO.txt file, and a `src/` directory. In general, all of our header (.h) files will be found in `include/` while all of our source (.c) files will be in `src/`. This is visually represented in Figure 1 on the left. Now, let's run **make** (just the command `make`, with nothing following that one word).

```
|-- GRADER_INFO.txt
|-- include/
|   |-- list.h
|   \-- s096.h
|-- Makefile
\-- src/
    |-- list.c
    \-- test.c
```

⇒

```
|-- GRADER_INFO.txt
|-- include
|   |-- list.h
|   \-- s096.h
|-- list-test.x
|-- Makefile
\-- src
    |-- list.c
    |-- list.o
    |-- test.c
    \-- test.o
```

Figure 1: Project directory structure before and after build

Notice there have been three files created: for every .c file, there's been a .o object file created. These files are linked together into the list-text.x program, which you can run. Some important points:

- **All of the implementation code you will write should go in the `list.c` file.**
- `GRADER_INFO.txt` is a file for the grader (don't edit!) containing the usual PROG: list, LANG: C
- **Important information for submitting your code:** you should submit a zipped folder using the same directory structure as the provided zip file. We've added a section in the Makefile for your convenience: if you type **make zip** in the same folder as your project, a zip file containing all of your code and the required headers will be constructed in the project directory and you can upload that to the grader.

We are provided a header file describing the interface for the **List** data structure. Three functions have already been defined for you in `list.c`: the code to create an empty list with no items: `List empty_list(void);` the code to destroy all the items in a list: `void list_clear( List *list );` and some code to print out a representation of a provided list: `void list_print( List list );`.

Look in the file `list.c` to find the functionality required of the other functions, which you will write.

```
#ifndef _6S096_LIST_H
#define _6S096_LIST_H

#include <stddef.h>

typedef struct List_node_s List_node;

struct List_s {
  size_t length;
  List_node *front;
};
typedef struct List_s List;

// Code you are provided
List empty_list( void );
void list_clear( List *list );
void list_print( List list );

// Code you will write
void list_append( List *list, int value );
void list_insert_before( List *list, int insert, int before );
void list_delete( List *list, int value );

void list_apply( List *list, int (*function_ptr)(int) );
int list_reduce( List list, int (*function_ptr)(int, int) );

#endif // _6S096_LIST_H
```

Some questions to ask yourself for understanding:

- How do you read the arguments to `list_apply` and `list_reduce`? Re-read this article.
- How is this a valid header to include when we don't ever define what the struct `List_node_s` really is?
- Why do we sometimes pass a pointer to a List struct and other times pass by value? When might we want to do it this way and when might we want to be more consistent?

Input Format

Not applicable; your library will be compiled into a testing suite, your implemented functions will be called by the program, and the behavior checked for correctness. For example, here is a potential test:

```
#include "list.h"
#include <stdio.h>

void test_print_list(void) {
  int N = 5;
  List list = empty_list();

  for( int i = 0; i < N; ++i ) {
    list_append( &list, i );
  }

  list_print( list );
  list_clear( &list );
  return 0;
}
```

Upon calling this function, the code outputs

```
{ 0 -> 1 -> 2 -> 3 -> 4 }
```

You are strongly encouraged to write your own tests in `test.cpp` so that you can try out your implementation code before submitting it to the online grader.

Output Format

Not applicable.

WARNING! FREE YOUR MEMORY!

This problem is only worth 300 points on the online submission system when you upload it. This is because **your code will be checked for memory leaks.** No memory leaks (a clean run in valgrind on our side) earns full points for a total of 400. Egregiously bad allocation, incorrect memory accesses, and lack of cleanup can result in as low as 300 points out of 400 maximum, even if all test cases are passing.