CSE 303, Spring 2007, Assignment 5B
Due: Monday 14 May, 9:00AM

Last updated: April 30

You will implement a "warehouse model" and unit tests for it. Other group members will *independently* develop a "unique-identifier data structure" and an "order-filling algorithm." The sample `warehouse.c` file is about 120 lines (this does *not* include other files). (Though the longest of the 3 assignments, the code has much easier algorithms.)

Requirements:

- Put your code in two files, `warehouse.c` and `warehouse_test.c`. Both should include `warehouse.h`, which you should write. Write an appropriate Makefile.
- `warehouse.h` (provided) should have just these prototypes plus typical header-file stuff:

```
#include "identifier.h" // also provided
struct Product;
struct Part;
struct Warehouse;

struct Warehouse * new_warehouse();
struct Part * add_part(struct Warehouse*, char*);
struct Product * add_product(struct Warehouse*, char*);
struct Part * get_part(struct Warehouse*, char*);
struct Product * get_product(struct Warehouse*, char*);
void add_part_to_product(struct Product*, struct Part*);
int product_count(struct Warehouse*);
int part_count(struct Warehouse*);
void receive_parts(struct Part*, int);
int sell_product(struct Product*);
```

- `warehouse.c` will use the declarations in `identifier.h`, so you will need to write *stub* definitions.
- In `warehouse.c`, define 5 structs (including two linked-list types) such that:
  - A *Part* has a pointer to an *ID* and an int *quantity* (the number currently available in the warehouse).
  - A *Product* has a pointer to an *ID* and a linked-list of *Parts* (those necessary to make the product; the same Part may be in the list multiple times if multiple are needed to make the product).
  - A *Warehouse* has two pointers to *IDSpaces* (one for Product IDs and one for Part IDs), a linked-list of all products, and a linked-list of all parts.
- `new_warehouse` returns a pointer to a new-heap allocated warehouse with no parts or products.
- If `add_part` is given a part-name that already exists in the Warehouse, it returns the `struct Part*` already in the Warehouse. (Hint: Use another function.) Else it creates a new Part, adds it to the list of all parts, and returns it. (Hint: You need to call `malloc` twice.) Use `string_to_id` and the IDSpace for Parts to get an ID. Intiailize the quantity to 0.
- `add_product` is like `add_part` except it returns a `struct Product*`, uses the IDSpace for Products, adds to the list of all products, and has an initial part-list of `NULL`.
- `get_part` returns the `struct Part*` in the Warehouse with the part-name passed as an argument (use `string_to_id` to get the right ID and then compare IDs with *pointer-equality*; it is up to the ID implementation to ensure this is correct). If no ID matches, return `NULL`.

- get_product is like get_part except it returns a struct Product*.
- add_part_to_product adds its second argument to the part-list of the first argument. (We assume both the Product and the Part are already in the same Warehouse.)
- product_count returns how many Products are in the warehouse.
- part_count returns how many Parts are in the warehouse.
- receive_parts increases the quantity of the Part it is passed by the amount of the int it is passed.
- sell_product updates the parts inventory for selling the Product. That is, for each Part in the part-list, we decrement its quantity. (If a Part appears multiple times, its quantity will decrement multiple times.) The return value is 1 if no Part's quantity becomes negative and 0 if some Part's quantity becomes negative.

Advice/Hints:

- Understand how all the pointers interact before you start coding. Be sure your struct definitions are right.
- Use the return value of sell_product for testing.
- You may also write a print_inventory function in warehouse.c to help with testing (and you will need it for homework 6 anyway).
- Some of the functions are very easy.
- Do not fret that you are not required to write memory-deallocation functions; this is to keep the assignment smaller.

Assessment and turn-in:
Your solutions should be:

- Correct C code that compiles without warnings using gcc -Wall and does not have space leaks
- In good style, including indentation and line breaks
- Of reasonable size

Your test code should provide good *coverage*.

Use turnin for course cse303 and project hw5. If you use late-days, use project hw5late1 (for 1 late day) or hw5late2 (for 2).