# Warehouse

## Project Description

For this project, we will be writing software to be used in a warehouse environment. The job in the warehouse is to assemble kits to send out to customers. Each kit is made up of several different kinds of widgets. Each different kind of widget is kept in a labeled bin in the warehouse. There are 100 different kinds of widgets in this warehouse, in bins labeled bin 0 through bin 99.

The parts that go into each kit are defined by orders that come into the warehouse. The orders consist of a list of bin numbers. Each bin number on an order indicates that you should take one part from the bin with that label, and put it in the kit. The same bin number may appear multiple times in an order, but you need to put the parts into the kit in the order specified on the order. For instance, if the order specifies "10 15 10 10 7 9 15 12", then the kit needs to have one part from bin 10, followed by one from bin 15, followed by two more from bin 10, followed by one from bin 7, one from bin 9, one from bin 15, and finally one from bin 8. Customers pay an average of 10 cents per part, so in this example kit, there are 8 parts, so the total revenue is $0.80.

But here is the trick. The workbench at which you assemble the kits can only hold 4 bins at a time. It starts out with no bins, but once you have filled all four slots on the workbench, if you need a bin that is not already on the workbench, you need to send a bin that IS on the workbench back to the warehouse, and fetch the new bin that you need. For instance, for the above order, you first need to fetch bin 10 and put it in the $0^{th}$ slot to fill the first order entry. Then fetch bin 15 to the first slot to fill the second order entry. You can fill the third and fourth order entries from bin 10 in the $0^{th}$ slot, but to fill the fifth entry, you need to fetch bin 7 to slot 2. The sixth entry requires a fetch of bin 9 to slot 3. Now, all the slots on the workbench are full. You can fill the seventh entry from bin

15 which is already in slot 1, but in order to satisfy the 8th order entry for a part from bin 12, you need to return one bin (say bin 10 from slot 0) and then fetch bin 12 into that slot. It costs 50 cents to fetch a bin, and another 25 cents to return a bin. In the above scenario, we needed to fetch five bins and return one so the total cost was $2.75 to assemble this kit, and the total revenue is only 80 cents, so the net revenue is -$1.95. We actually lose money on this simple kit.

We make money in our warehouse because most kits consist of the same parts over and over again. In fact, the probability is very high that the next part that goes into the kit is a part that we retrieved for the kit recently. Your job is, knowing that most kits use the same parts, to figure out how to best use the slots on the workbench so that you can maximize your profit.

## Working on the Project

You have been provided with the basic infrastructure for the C code to simulate the warehouse described above. On the project sub-page of the class web page, there is a file called proj1.tar.gz that you can download to your own UNIX directory. The command:

```
tar –xvzf proj2.tar.gz
```

will first create a sub-directory of your current directory called “proj1”, and then populate that sub-directory with the contents of the proj1.tar.gz file. The proj1 sub-directory will contain the following files:

- warehouse.c – C source code that contains the main function which simulates the warehouse. This is the file and function that you need to modify. The main function, as it is delivered, performs the following functions:
  - includes slots.h. This enables access to the utility functions described in slots.c below.
  - Invokes the “initSlots” function
  - As long as there are more bin numbers on the order for the kit, main will:
    - Read the next entry from standard input, and save it in the “bin” variable.
    - Check to see if that bin is already on the workbench by invoking the “findSlot” function. findSlot will return a -1 if the bin is not already in a slot on the workbench.
    - If bin is not already on the workbench, decide which slot to put the bin into on the workbench, and invoke the “getBin” function which retrieves the bin from the warehouse, and puts it in the specified slot on the workbench.
    - Invoke the “getWidget” function to remove a part from the specified bin (which must now be in a slot on the workbench) and put it in the kit.
  - When the order is complete, invokes the printEarnings function to print out the total cost and revenue for this kit.
- slots.h – A “header” file that enables warehouse.c to invoke the functions defines in slots.c.

- slots.c – This C source code contain the functions which simulate the workings of the warehouse. You may not modify this code or these functions. However, you may invoke these functions, as needed. The functions in slots.c are as follows:
    - void initSlots() – This function initializes all slots on the workbench to "empty" (bin=-1)
    - void getBin(int bin,int slot) – This function checks the specified slot. If it is not empty, the bin in that slot is returned to the warehouse, and $0.25 is added to the cost. Then, the specified bin is retrieved from the warehouse and placed in the slot specified, and $0.50 is added to the cost. Finally, the getBin function prints out a line that indicates which bins are in each slot.
    - int findSlot(int bin) – This function checks through all the slots to figure out what slot the specified bin is in. If the specified bin is in a slot on the workbench, the findSlot routine returns the slot in which that bin resides (0, 1, 2, or 3). If the bin is not on the workbench, findSlot returns a -1.
    - void getWidget(int bin) – This function checks to make sure that the specified bin is in a valid slot on the workbench. If so, indicates that you have added a part from that bin to the kit (increasing the value of the kit by $0.10)
    - void printEarnings() – Prints a line to standard output that identifies the cost and the net revenue for this kit.
- Makefile – a make file that contains several targets, as follows:
    - test : A pseudo-target to invoke the warehouse executable file, which redirects "order1.txt" to standard input.
    - Debug: A pseudo-target to in
    - warehouse: Creates the warehouse executable file using both the code in warehouse.c and the code in slots.c
    - clean : A pseudo-target to remove the warehouse executable file, and the submission tar file
    - submit : A pseudo-target to create the tar file to submit on blackboard. NOTE: If you do not use "make submit" to create the tar file, you will get points deducted for not following directions.
- order1.txt – A very simple list of bins, as specified in the example above.
- order2.txt – A slightly more complicated kit order… one which we can actually make money on if we are smart.

When you first untar the file, cd to the proj1 directory, and try building and running warehouse. Note that as delivered, the "main" function uses only slot 0 on the workbench. That means that there is a lot of schlepping bins back and forth to the warehouse, and it's hard to make any money on an order.

Your job for this project is to modify the main function so that it chooses a bin to return (and free up a slot for a new bin) intelligently. The trick is to send back the bin that is least likely to

be needed again in the near future. If you can do so, then you can reduce the cost of kits, and increase your profit (or decrease your loss).

You may add new functions and / or variables to warehouse.c, and change the code in the main function. You may not "look ahead" in the order list… you must fill each order entry as it arrives, before looking at the next order entry. You may not modify the functions in slots.c (which keeps track of the cost and benefit of packing a kit.) I have given two sample orders in the file "order1.txt" and "order2.txt". Your program should run with any valid order (arbitrary list of numbers between 0 and 99).

## Standard Input, Standard Output, and Standard Error

We have not yet talked about the "standard" input and output (IO) streams in C, but this project makes use of those streams. Most of the interaction with these standard IO streams has been provided to you in the infrastructure, but it's worth describing them in slightly more detail. For more detailed information, see https://en.wikipedia.org/wiki/C_file_input/output, or look at The C Programming Language (Kernighan and Ritchie) chapter 7

In UNIX and in C, every program has one input IO connection called "standard input", and two output connections called "standard output" and "standard error". In C, IO (including file IO) is handled by a concept called a "stream". In C, the three standard IO connections are all streams; the standard input stream, standard output stream, and standard error stream.

Normally, C connects the standard input stream up to your keyboard. With that connection, when your program requests input from standard input, then your terminal opens up (the cursor blinks), and the program waits for you to type something on the keyboard. Whatever you type doesn't get to your program until you hit the "Enter" key. When hit the "Enter" key, what you typed goes into the standard input stream, and is available to be consumed by your program. If you type "Ctrl-D", that sends an "End of File" signal to your program. (Note that a "Ctrl-C" key sends an immediate "kill" signal to your program. You can use this if your program ends up in an endless loop.)

For the purposes of this project, we use stdin to provide the order for a kit; a list of bins. UNIX supports redirection – the capability to get connect standard input to a disk file rather than the terminal – by using a less than (<) sign. Therefore, the command "./warehouse <order1.txt" invokes the warehouse executable, and sends the "order1.txt" file in the current directory to standard input. That allows "order1.txt" to contain the list of bins which contain the parts for kit1. To make kit2, we would specify "./warehouse <order2.txt".

## Academic Honesty

It is not very likely that you will be able to copy and paste code from the internet into your code… any code from the internet probably will not be compatible with your infrastructure. Usually, it's much more effective, and definitely a much better learning experience if you write your own code from scratch.

Feel free to discuss this project with other students. However, ***DO NOT COPY CODE!*** Your code will be compared with all other students' code. The compare tool looks at the semantics of the code – not just the bytes, so it can detect copied code even if you add comments, change variable names, etc. If your code compares too closely with any other student's code, then both the copier and the student who wrote the code that got copied will get a zero grade on this project. If you use the same concepts, but write your own implementation of the code, there will be enough of a difference in the result so that you will not be accused of cheating.

## Submitting your Code

When you have finished coding and testing for this project, run:

```
make submit
```

in your project directory. This will collect your source code and put it in a tar file called "proj1_<userid>.tar.gz", where "<userid>" is your LDAP user ID. (Note that your userid is encoded in tar file itself as well as in the file name, so just renaming the tar file is not good enough. You MUST run make submit on an LDAP machine, or "make USER=<userid> submit".) Then upload the proj1_<userid>.tar.gz file onto blackboard in the Project 1 submission area. Note that you may submit as many times as you want. The TA's will only grade the latest submission.

## Project 1 Grading

Project 1 is worth a total of 100 points. After the due date, your proj1_<userid>.tar.gz file will be downloaded onto an LDAP machine, untarred, and compiled using a Makefile similar to the one you have been using. You will start with 100 points.

- There will be a 10-point deduction for each 24 hours your project is late up to a maximum of 5 days. (After 5 days, you will receive a zero.)
- If you failed to follow directions, and upload a tar other than one created by running make submit on an LDAP machine, you will get a 15-point deduction.
- If your code gets a compiler error, the TA or professor will try to fix your code. If there is a simple fix, we will make that fix, subtract 20 points, and continue testing. If there is no simple fix, you will get a grade of 20 for this project.
- Once you code compiles, 10 points will be subtracted for each type of compiler warning in your code.

Then, your program will be run on order1.txt and order2.txt, as well as three unpublished test cases; order3.txt, order4.txt, and order5.txt

- If your code does not fill an order, no matter what the resulting net earnings is, you will get a 15-point deduction per unfilled order.
- Your net earnings will be compared to the maximum possible net earnings for each order. If your earnings are less than the maximum, you will get a deduction of 10 times the percentage you underproduced, using the formula: $10\times {Your\_earnings}/{Max}$

Note: This is the first of four projects, and your lowest project grade will get dropped. So don't despair if you don't do well on this project… that means you will just have to work harder on projects 2, 3, and 4.