Skip to page content

Lab 1b: Data Representation in C

Assigned: Thursday, April 4, 2024
Due Date: *Monday, April 15, 2024 at 11:59 pm*
Video(s): Aisle Manager, Store Client, and Checking Your Work

Overview

Learning Objectives:

- Extract useful information from data using bitmasks and bitwise operators.
- Recognize differences in C data types and use C casting.
- Explain the relationship between pointers and arrays in C and use it to access and manipulate data.

You will interact with a custom data representation for keeping track of things in a store, implementing functions using bitwise operators, pointers, and arrays.

Code for this lab

Browser: Download here
Terminal: `wget https://courses.cs.washington.edu/courses/cse351/24sp/labs/lab1b.tar.gz`
Unzip: Running `tar xzvf lab1b.tar.gz` from the terminal will extract the lab files to a directory called `lab1b`.

Lab 1b Instructions

Lab Format

This lab uses a custom data representation of aisles in a store. `aisle_manager.c` and `store_client.c` have skeleton functions for you to implement, each with a comment describing its desired behavior.

You will start by implementing functions that use bitwise operators, bitmasks, and C control structures to manage an individual **aisle**. After implementing these functions, you will move onto functions that use pointers, arrays, and your aisle functions to manage multiple aisles in **a store**. We have provided two test programs to help you check the correctness of your work (see Checking Your Work for more details).

As you work through the lab, you are encouraged to test each function one-by-one as you go (see Checking Your Work for instructions on how to just test one function). *We also encourage you to use previously implemented functions and previously defined macros to help you solve future functions!* Please post on the message board if you are unsure whether or not it is ok to use something.

Style

This isn't a programming course and we generally won't be checking your style. However, for this lab we will do a quick check for the following items, with the intent of helping you avoid introducing unnecessary bugs. This will be worth just 1 point and is intended to be relatively stress-free.

- Magic numbers: You should not hard-code constants in your code (exceptions: `0`, `1`) when you can define and (re)use macros to give them meaningful names and consistent values.
- Function usage: You should use defined functions when appropriate, especially the `get_*` and `set_*` functions.

Aisle Manager

Start by reading the description of an aisle's data representation at the top of `aisle_manager.c`. ***Make sure you fully understand the layout of the representation before continuing. If you have any questions, please post on the message board or come to office hours.***

Once you understand the data representation, you can begin to implement the skeleton functions in `aisle_manager.c`:

- `get_section`
- `get_spaces`
- `get_id`
- `set_section`

- `set_spaces`
- `set_id`
- `toggle_space`
- `num_items`
- `add_items`
- `remove_items`
- `rotate_items_left`
- `rotate_items_right`

Do not be discouraged by the number of functions! They build gradually in level of difficulty, and many of them are similar to each other. Here's one recommended way of working through the file:

1. Start by filling in the macros at the top of the file with various bitmasks to be used later.
2. Continue by implementing the `get_*` and `set_*` functions.
3. Finish by implementing the rest of the functions related to manipulating the items in an aisle.

At this point you should be able to pass all tests in the `aisle_test` executable! Now you can move onto `store_client.c`.

Store Client

`store_client.c` has some functions for interacting with a group of aisles and a stockroom in the store. Pay particular attention to its use of global arrays: (1) of type `unsigned long` to represent multiple aisles and (2) of type `int` to represent a stockroom for the store (items stowed away that are not in the aisles).

We have not talked about C global variables in depth, but for the purpose of this assignment, you can treat them as you would global variables in Java. That is, you can use the global variables in any of the functions as though they were local variables without declaring them as local variables. There are slight differences between global variables in C and Java, but they are not relevant for this assignment.

Skeleton functions:

- `refill_from_stockroom`
- `fulfill_order`
- `empty_section_with_id`
- `section_with_most_items`

At this point you should be able to pass all tests in the `store_test` executable! This means you are done with the programming part of the assignment (don't forget about the synthesis questions!).

Checking Your Work

We have included the following tools to help you check the correctness of your work:

- The functions `print_binary_long` and `print_binary_short` take an `unsigned long` and `unsigned short`, respectively, and output its binary representation. These are optional to use but can be useful in debugging your code. **All calls to these functions should be removed from your final submission.** Here is an example usage of `print_binary_long` (`print_binary_short` works similarly except it takes an `unsigned short` as its argument):

```
void print_binary_example(unsigned long* argument) {
  // Print out binary view of argument for debugging.
  // The type of argument is unsigned long*, so we
  // must pass its dereference to print_binary_long.
  print_binary_long(*argument);
}
```

- You can also use `printf` to debug your functions, but you'll have to add `#include <stdio.h>` at the top of your file. **All calls to `printf` should be removed from your final submission.**
- `aisle_test` and `store_test` are programs that ***check the functional correctness of the code*** in `aisle_manager.c` and `store_client.c`, respectively.

  You will need to rebuild these executables before running them *each time you modify your code* using the `make` command.

  To test all of the functions, use the following commands:

```
$ ./aisle_test
$ ./store_test
```

  The test one function at a time, use the `-f` flag on the appropriate executable, *e.g.*:

```
$ ./aisle_test -f set_id
$ ./store_test -f section_with_most_items
```

  - `aisle_test` and `store_test` first test your solution on specific cases, and then on a wide range of inputs. As a manner of limiting the test output, for the range of inputs the testing program will only print out the first input that your solutions are incorrect for (or no input if you pass all cases).
  - There is some partial credit awarded if your function passes the specific cases but not the range of cases, and each function is tested separately to allow for the opportunity for partial credit should you only complete some of the functions.

Lab 1b Synthesis Questions

Make sure your answers to these questions are included in the file `lab1Bsynthesis.txt` !

1. Data representation: Your friend suggests an alternative data representation for sections in an aisle. In this representation, the lowest 8 bits would use unsigned integer encoding to represent the number of items in a section, and the upper 8 bits would be used to store a unique item id. [4 pt]
   - Compared to our current representation, give **two** benefits and **one** drawback of this new representation.
   - If you were the store manager, which representation would you prefer to use? Briefly justify your choice.
2. Number representation: Consider the following two statements (noting that the type of `y` is purposefully omitted):
   - `y = -1;`
   - `y = 0xFFFFFFFF;`

   Is there a difference between using these two statements in your code? Explain. If there is a difference, make sure to provide an example. [2 pt]
3. Floating point: Explain why comparing two floating point numbers with == or != is problematic. The suggested equality alternative from lecture involved comparing the difference against a threshold value: what should be considered in choosing such a threshold? [3 pt]

Submission

You will submit: `aisle_manager.c` , `store_client.c` , and `lab1Bsynthesis.txt` .

Be sure to run `make clean` and then `make` on your code before submitting! Submissions that don't compile will automatically receive a score of ZERO.

After submitting, please wait until the autograder is done running and double-check that you passed the "File Check" and "Compilation and Execution Issues" tests. If either test returns a score of -1, be sure to read the output and fix any problems before resubmitting. Failure to do so will result in a programming score of ZERO for the lab.

Submit your files to the "Lab 1b" assignment on Gradescope. Don't forget to add your partner, if you have one.

It is fine to submit multiple times up until the deadline; we will only grade your last submission. If you do re-submit, you must re-submit ALL files again AND add your partner again to the new submission.