CS1010 Laboratory 10

Struct, Wrap-Up

Zhang Puyu

Group BD04

14 November , 2024

Plan of the Day

1. Exercise 8 Review

2. Preamble to OOP: `struct`

3. Anything Else You Want to Talk about

Exercise 8

- This is quite a tedious problem but it is not that difficult.
- We want to retrieve the k nearest training samples given a testing sample.
- This is the same as retrieving the k smallest numbers from a list. Only difference is the way to calculate the near-ness:
    - Let S and T be the sets of `'#'`'s in a testing sample and a training sample.
    - Their "distance" is defined as the number of pixels **in one sample but not the other**.
    - Basically, $|S \cup T| - |S \cap T|$ (in technical terms: symmetric difference).
    - Add-on: in real scientific research, the more standard way to represent the similarity between two images is **intersection over union**.

Preamble to OOP: `struct`

So far, we have been representing different objects using **primitive types:** `int`, `float`. (Note that `bool` and `char` are essentially integers.)

In real life, many objects cannot be easily mapped to a single variable.

Example: Personal information consists of

- name (`char *`)
- age (`long`)
- home address (`char *`)
- birthday (`long *`)
- family members (arrays of the above items)

Preamble to OOP: `struct`

- name (`char *`)
- age (`long`)
- home address (`char *`)
- birthday (`long[3]`)
- family members (arrays of the above items)

How to represent a list of persons then?

- `char **name`: an array of names
- `long *age`: an array of ages
- `char **address`: an array of addresses
- `long (*birthday)[3]`: an array of birthdays
- `long **family`: `family[i][j]` is the *j*-th family member of the *i*-th person

Preamble to OOP: `struct`

- `char **name`: an array of names
- `long *age`: an array of ages
- `char **address`: an array of addresses
- `long (*birthday)[3]`: an array of birthdays
- `long **family`: `family[i][j]` is the *j*-th family member of the *i*-th person

This is **verbose** and **unintuitive!**

Suppose we now have a **composite data type**, `person` that captures all the information, then we will only need an array of `person`!

This is the motivation to use a `struct`.

Preamble to OOP: struct

```
struct person {
    char *name;
    long age;
    char *address;
    long birthday[3];
    // Use pointer as we want to access
    // the actual person instead of a copy
    person *spouse;
};
person *list = read_persons();
```

Now the syntax is much clearer and readable! In fact, we could define another *struct* called `date` to represent the birthday!

In higher-level languages, this evolves into what is known as **Object-Oriented Programming** (taught in detail in CS2030/S and CS2113(T), and somewhat needed for CS2040/S/C).

Moving Forward from CS1010...

Let's start with an (personal) assertion:

> ***The more you think you know about C, the less you actually know about it...***

Things to look forward to after CS1010:

- Learning other programming languages:
    - C++ (officially taught in CS2040C)
    - Java (officially taught in CS2113(T) or CS2030(S))
    - Python? Needed for data analysis and machine learning.
    - JavaScript/TypeScript? You need it for web development.
    - C# ? An ideal choice for solo development (personally highly recommended).
    - Others: Go, Rust, Swift, Scala, etc.
- Learning about other programming paradigms.
- Learning about software engineering principles and design patterns (officially taught in CS2113(T)).
- Doing you own projects!

What to Expect in CS2040C?

- Fundamentals of C++ and OOP.
- Data structures, their implementations and applications:
  - Arrays and Lists
  - Linked Lists, Stacks and Queues
  - Heaps and Priority Queues
  - Hash Tables
  - Binary Search Trees
  - Graph representations
- Basic-to-intermediate algorithms:
  - Sorting Algorithms
    - Insertion, selection, bubble, counting
    - Merge sort, (randomised) quick sort, heap sort, radix sort
  - Graph traversal
  - Single-source shortest path
  - Minimum spanning tree

Introduction to Different Programming Paradigms

A “programming paradigm” is a framework for solving computational problems. Suppose we have a problem as follows:

- We are given a list A of integer triplets $A[i] := (a_i, b_i, c_i)$.
- We want to keep only triplets with all-positive entries.
- We want to **group** the triplets based on their first entry.
- We want to find the group where the sum of the second entries over all triplets in the group is the smallest.
- We want to output an integer pair (M, S), where M is the value of the first entry corresponding to the group, and S is the sum of the third entries over all triplets in the group.

How would a C programmer (that is, you) approach this problem? How would a programmer using a different language reason about the solution differently?

The Procedure-Oriented Way of Thinking

- Write a function `get_result`
- `get_result` needs to know the first entry of the minimum group.
- Write a function `get_min_group` for that.
- `get_min_group` needs to know the first entry of all groups.
- Write a function `get_groupings` for that.
- Before running `get_groupings`, we need to filter out triplets containing non-positive entries.
- Write a function `filter` for that.

As the task grows in complexity, the number of helper functions increases drastically, which makes our program very hard to maintain.

The Object-Oriented Way of Thinking

- Suppose we have an object called **List**.
- Let's define a **functionality** exclusive to the list called `keepOnlyPositive`.
- Let's also have another object called **Group** which is essentially a labelled list.
- Now, we define another functionality for our list called `groupByFirst`.
- In our group object, we can define a `sum` functionality.
- In our list object, let's have a third functionality called `maxBy` which will find the "maximum" according to an attribute (which we will supply as the sum over each group).
- Now we can invoke the functionalities in the correct object and the correct order to get the desired output.

We consider the objects needed to solve a problem, and define the functionalities of each type of objects according to what behaviours or interactions we wish them to support.

The Functional Way of Thinking

- At a higher level of abstraction, what really happens underneath is just a randomly generated list **undergoing a sequence of transformations**.
- Let's instead forget about the objects and just focus on the transformations.
- Define f_1 as the mapping from a list of triplets to a list of positive triplets.
- Define f_2 as the mapping from a list of triplets to a list of grouped triplets based on their first entry.
- Define f_3 as the mapping from a list of grouped triplets to the group with the minimum sum of the second entries.
- Define f_4 as the mapping from a group of triplets to a list of triplets.
- Define f_5 as the mapping from a list of triplets to the sum of the third entries.
- Define f_6 as the mapping from a group to its label.
- Then, we can find the "minimum group" with $G \coloneqq L \to f_1 \to f_2 \to f_3$, then $M = f_6\left(G\right)$ and $S = G \to f_4 \to f_5$.

The functional paradigm views a problem-solving process as a **pipeline** of function applications. By carefully crafting the functions, we can pass in the initial argument into this pipeline and collect the result at the other end.

Other Stuff for Remaining Time

- Go through PE2
- Clear doubts about any concept taught over the semester
- Suggestions for life as a programmer after CS1010
- Others