Project 2B - Rank in C

Group G

Josef Dewberry
CSC 417
North Carolina State University
Raleigh, NC, USA
jadewber@ncsu.edu

Thea Wall
CSC 417
North Carolina State University
Raleigh, NC, USA
adwall4@ncsu.edu

Spencer Yoder
CSC 417
North Carolina State University
Raleigh, NC, USA
smyoder@ncsu.edu

1. Introduction

Pipeline: rank
Language: C
Abstractions: Pipe & Filter
Tantrum
Quarantine

2. Abstractions

2.1 Pipe & Filter

A pipe and filter is meant to simplify the flow and organization or code in a program. The filters are sections of code that transform data, and the pipes pass on the transformed data from one filter to the next

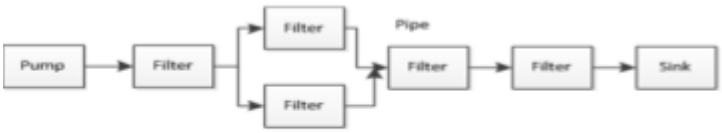

Figure 1 - Visual representation of a pipe and filter provided by https://paramore.readthedocs.io/en/latest/BuildingAPipeline.html

These are best used when a large chunk of data being edited, and the steps of editing can be broken into different sections (i.e. a spreadsheet that must first remove unnecessary data, then take the remaining data to construct a new set of data, and then finally adding that set of data to the spreadsheet).

This also allows multiple programmers or teams of programmers to work on different parts of a program at once, as each filter should NOT change the functionality of the other filters. A filter should be a stand alone piece of code. If parts of a program must affect other parts of how the program functions, a pipe and filter is not a good idea.

Pipe & filter would work well for rank because rank goes through several different steps. It first must divide the input into usable objects, sort through those objects different fields in order to rank the input by the best, and then separate those rankings into output. Having each of those steps as a filter would be perfect.

2.2 Iterator

An iterator mainly deals with lists and collections of data. Every collection of data must have some sort of ability to go through every object in the collection, otherwise it is simply a tally of objects. An iterator does just that.

Generally an iterator should not expose the type of collection it is bound to. Iterators should act the same for all collections so functionality in code can be reused and the developer doesn't have to constantly worry about the type of collection they are using.

Iterators are generally very useful. The few shortcomings they have are generally because of the implementation. Collections usually cannot be modified while an iterator is in use, as it can mess with the iterator's position in the collection. They will also add to the complexity of a program, sometimes in a big way depending on the collection and the iterator's implementation.

An iterator would be highly useful for this program. A main point to rank is taking a large input and breaking it into usable data types. If those data types have a way to easily move through collections of themselves, then working with the data would be a breeze.

2.3 FlyWeight

The flyweight pattern is intended to keep the number of objects in a program to a minimum. This is done by keeping a collection of objects collected. If the code attempts to create an object, the collection checks if that object type has been created, and if it has it returns a reference to the already created object. If not then it creates a new object and returns that instead.

The main advantage of using the flyweight pattern is reducing memory. In any program where memory is limited, sparing just a little memory for the collection of objects can save a lot of memory of redundancy in objects. Object memory is shared, meaning memory used goes down.

The downside of using flyweight is that the program has traded memory for CPU time. Time spent searching through the collection for an already created object could have been spent simply creating the object. Time vs. memory is a constant struggle in any program. Using flyweight also means that the code becomes more complex by a good deal. Constantly changing referenced objects leaves a lot to keep track of.

FlyWeight would be easy enough to implement in C. We would create a structure that keeps track of objects used and call that structure as needed. The problem is the constraints we are working under don't require flyweight. Flyweight sacrifices time for memory, but neither of those are really an issue. The program running successfully is all that matters. Implementing FlyWeight seems unnecessary, but still doable..

2.4 Recursion

Recursion is the act of calling a method or other piece of code within that very method or piece of code, thereby repeating itself until the code reaches a solution.

Recursion requires both base cases and recursive cases. A base case is when a function can give an answer without calling itself. Usually a base case is very simple, but not always. The recursive case is when the function needs to recursively call itself. Generally the input given will be broken down, and then one or more of the pieces of input will be used to make the recursive call.

Recursion is a great way of keeping code simple and clean. Having one recursive function instead of multiple functions that do the same thing makes reading the code easy. If the input must be traced however, it can get pretty complex. Tracking input through many recursive calls while also keeping track of the splintered inputs can be quite difficult.

Recursion is very simple in C. A functions just has to call itself. Rank could recursively call itself as it sorts through the input file, although this would probably be better dealt with via a data structure rather than recursion. Rank has to do a lot of different things to the input, such as sort through it, pick out the different independent variables, etc.

2.5 Layers

Layering is a technique in which code is built into sections in which it can only communicate with two other pieces of code, the code "above" it and the code "below" it,

effectively creating layers. Much like the earth's layers, each layer is only touching the layer above and below it.

Layering can be seen in the development of operating systems. The bottom layer would be the hardware, and the highest layer would be the user interface. Each layer in between serves as a step to get from the user's request down to the hardware which can actually make the necessary calculations to carry out the request.

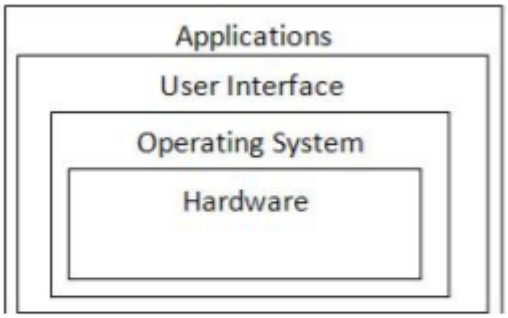

Figure 2 - A visual representation of a layered OS, provided by https://en.wikibooks.org/wiki/IB_Computer_Science/Science_Systems_Life_Cycle_and_Software_Development/Systems_Design

The main advantage of layering is the ease in which a developer can construct a layered program. Layering allows for each part of the program to be done in a well defined step. The developer knows exactly where the code just came from, and where the code will go next. It also helps with debugging, as the developer just has to search for what layer the bug is in. The drawback of layering is it limits functionality. If it would be beneficial for one layer to communicate with a layer two layers up, or for a layer to change how another layer works, that is simply impossible.

Layering would be easy to implement in C as you just have different functions only talk to certain functions just a layer would. Rank doesn't really work for layers though. Rank works more in a streamlined way, starting with the input and constantly moving for the output. Each layer would have no reason to talk to the previous one. The layering would look less like layers and more like a slide.

2.6 Tantrum

Tantrum is the constant checking for errors in the code. Every argument should be checked to make sure it is valid, and every possible error should be checked along the way. If an error is found, deal with it accordingly. The developer can have the program print out an error specific message, or pass the error up the function call chain.

The use of tantrum is dependent on the programming language being used. If the language being used is one that doesn't automatically check inputs, data types being used, etc., then tantrum can be quite useful. If the language does automatically do that kind of error checking, then rechecking the issues is moot. Tantrum also is dependent on the program being written. A program that deals with life or death situations (e.g. hospital equipment, airline software) should absolutely notify the user if something has gone wrong. If the program is a video game, sometimes bugs aren't the end of the world and can be worked around.

For the use for our program, tantrum would be great because we are using C. C allows the user to do a lot of things another language like java wouldn't because it trusts the user. Also because rank deals with sorting through lots of input, if something were to go wrong our output would be terrible off and we wouldn't know until we view the output.

Tantrum might be a bit difficult for us to implement because of the sheer amount of things rank has to do. Implementing tantrum would require implementing a lot of error checking, because each step has a lot that could go wrong.

2.7 Factory

A factory is a method or piece of code that creates different types of classes or objects. These classes or objects generally have a shared base class or interface.

Factories are incredibly useful in situations in which many different types of

classes or objects are needed on a case by case basis. Instead of the developer having to constantly create new objects or call constructors, the process is automated.

Factories allow the developer to more easily change the code if needed. The change must only be made to the factory. They can be detrimental though by muddying up code. An improperly used factory will make the code less efficient and harder to read than if the developer had just created objects and classes with constructors instead.

Rank as a program does not really lend itself to needing a factory. Simple functions and pre-made data structures would work fine for breaking the input into usable data. C as a language lends itself very well to make factories though. With structs, struct pointers, and function pointers we could easily implement a factory.

2.8 Map Reduce

Map reduction is useful in situations that deal with a large amount of data. Functions that do the same thing are used to work on different parts of the data. This allows the program to do virtually the same processes on different parts of data simultaneously

Figure 3 - A visual representation of MapReduce, from https://developerzen.com/introduction-to-mapreduce-for-net-developers-1030e070998e

This speeds up production on batches of big data, but the programmer must keep the weight of processing speed vs. computing power. The act of map reduction is heavily taxing, as it requires multiple threads to be at work. If the amount of data being worked on is not large enough, then the strain on the computing power is unnecessary. Also, if time is not a big factor but rather computing power is, then map reduction may not just be unnecessary, but rather impossible.

Map reduction would be very useful for Rank. Rank deals with large amounts of input and must edit it in several different ways. The biggest problem is the complexity vs. payoff. Implementing map reduction would take a lot of work, and our project doesn't have neither memory nor time constraints. If the program had to run under a certain amount of time, it would be useful. But as of now, the only benefit to map reduction would be to make the program seem cool.

2.9 Macros

Macros are rules that allow for a sequence of code or data to be transformed to a different sequence of code or data. This is done through a macro expansion, which is a specific sequence that instructs the language to do the remapping in the specified ways.

A very simple but easy to understand example of macros is the use of #define in C. #define is commonly used to create global variables (e.g. #define PI 3.14 would change all instances of PI in the code to 3.14, which is a usable double data type). #define can also be used to set up quick functions (e.g. #define double(x) ((x) * (x)) would take any instance of double() in the code, and multiply whatever x is by itself. Therefore double(3) would return 6).

Macros can make code easily changeable, as the point of a macro is to be used more than once. If something in the macro must be changed, simply change the macro instead of changing multiple instances of code. Macros do come with the disadvantage of making the code less readable and more difficult to debug. Finding an issue in a code with frequent macro usage means having to recheck a macro several times when following input through the code.

Macros would be easy to implement in C, as already demonstrated in the example. The problem is implementing them in a worthwhile way. Anyone can make global variables as a "macro" but to really show off the power of a macro we would need large amounts of code that seemingly does the same thing. That could

probably be done with how the code searches through the input, but that could also easily be done just once and then the usable input is saved into a data structure. Also code readability is highly important, and using macros would hurt that.

2.10 Quarantine

The quarantine pattern involves separating IO from the core program functions. It forces the program functions to be more “pure” in the sense that calls to any of them should always return the same value without side effects. This can be done by wrapping functions which perform IO in higher-order functions which are only executed in the main program.

This makes code more predictable, easier to reason about, and easier to test. The pattern forces programmers to think carefully about what functions do IO. However, programmers are not necessarily forced to minimize IO, and it can add complexity to the program.

When implementing the rank program in C, this pattern might be useful for isolating memory allocation to avoid problems with memory leaks.

3. EPILOGUE

The story of this particular programming project was one of overcoming problems and roadblocks because of the constraints we imposed upon ourselves. Firstly, programming in C proved to be much more of a challenge than programming in Java. C requires the programmer to do their own memory management and has incredibly limited support for working with strings. This made reading in and processing lines of input much more difficult than Java. What made C a good language to work with was the ability to chain function calls to facilitate the pipe/filter abstraction, the ease of separating code into discrete functions, and the ability to store data in structs.

The pipe/filter abstraction, out of the three that our group implemented, was probably the easiest. All this required was a series of chained function calls with information being passed between them. The transformative nature of rank made this a relatively straightforward process, though there were some snags. For example, there were times where it would have been convenient to return more than one object or data structure, like the length of a list along with the list. This could have been worked around with either structs or global variables, but we set out with the goal of minimizing the declarations of these things.

The quarantine pattern was relatively trivial to implement, but did involve an unfortunate waste of memory and efficiency. Our original implementation involved reading lines of input in two functions, which is not allowed under quarantine. This was remedied by reading every line from standard input into a data structure and then getting lines of input from this data structure. Quarantine and pipe are relatively compatible, our problem was a lack of planning.

Tantrum was certainly the most tedious to implement. Especially in C. We were aware of the ability to throw messages up the stack in C, but could not recall how to implement this. So not only did we have to try and consider/write in conditions for every error case, we had to pass them up the stack without the use of these tools. Luckily, because of our pipe and filter implementation, we had a chain of function calls, each returning something. The way we approximated a stack trace was printing a contextual error message every time a function received something it didn’t expect.

4. MAXIMUM GRADE EXPECTED

Base Points: 10
Pipe Choice: 1 (Rank)
Language: 0 (C)
Abstraction: 5 (Pipe & Filter, Tantrum, and Quarantine)

The maximum points expected is 16.