CS152/154

Shubh Kumar

IIT-B,Spring Semester 2021

1 Lecture 1: Review of Program Design

- **Conceptualization:** The theorems, proofs, Design, suitability, satisfiability
- When Software began becoming complex, only then people understood the importance of Program Design.
- **Re-usability:** Making Programs like Black-Box like Components which can be simply put somewhere else, this is done by making the program standardized and having it adhere to specifications. We need to structure our programs so that many of these form a particular structure which can be reused somewhere else, or can be put from somewhere else.
- **Data and Control Abstractions:** Flow of data and the abstractions which allow us to change the data/state comes under Control Abstractions. The way we store this data could be comes under Data Abstractions.
- Control in real life and thus in programs have many branches through which you want to change the state.
- Boolean, Integers, stuff like this come under Data Abstractions, there are these basic data abstractions, using which we could structure more complex data abstractions.
- **Programming with Assertions:**Asserting gives us defences against some mistakes we may have unknowingly made in the program. We add assert statements which terminate the program just as soon as the assert conditions goes false
- **Reasoning about Programs: Proving correctness** How to make sure that what we are doing is correct
- **Imperative Programming Ideas:** Ideas revolving around the change of states as the basis of Programming
- **Applicative Programming Ideas:** Ideas around functions operating on arguments as the basis of programming
- **Procedural Paradigm:** The way its done in C,C++, FORTRAN i.e. We have some procedures and global variables.
- **Object-Oriented Programming Ideas:** Ideas revolving around packaging into objects

2 Lecture 2 : Monday's Lab Discussion

- The C program was supposed to be procedural, whereas the C++ program needs to be Object-oriented
- A Good Programming Practice is to declare all the variables in the beginning of the main function/file.
- Commenting usage/documentation in the beginning is another great programming habit.
- **C++ Resource:** Click Here

- **Linux Commands:**
  - man command in Linux: To Find documentation
  - mv command in Linux: To Move files from one place to a folder
  - For more basic commands Go here.
- srand() sets the seed, which is basically an attribute for a beautiful random number generator, if we have the same seed or set no seed, then it'll give the same seed no matter how many times we run. Thus, We should almost always use srand() and a seed whenever we use rand()

3 Lecture 3: Critical Discussion on Monday's Lab and Programming Discipline

- `#include<...>` includes header files, which are links to other pre-defined libraries.
- `using namespace ...` includes a particular set of functions(in namespace, ie the space of all possible names) which identify with a particular library/task named in ...
- **Proper Spacing:** `int main () {}` is better than `int main(){}` , but not too many spaces and also like `cout << "hello\n";` instead of `cout<<"hello\n";`
- `printf(10)` and similar would give error, as the first argument is supposed to be only the format, and only in the placeholders could numbers be added.
- If we try to print something without initialization, it prints garbage values.
- For `printf()` and `scanf()` syntax properly Click Here.

4 Lecture 4: Abstract Data Types

- We'll be requiring `libfltk1.3` and `fluid`(part of Fast Light Toolkit), for graphics as the course progresses.
- When there's an interaction between different variables, Global Variables become useful, although they do have their drawbacks.
- The expressions in Programs are of two kinds, Data and Control.
- Data represents the state Problem Domain, and Control represents Data Manipulation to get various results.
- Compilers give us the feature to construct newer Data Abstractions, and overloading the operators on the pre-existing data types and stuff.
- Even the primitive Data Types have arosen from some abstractions, abstracting meaning from the way bits are set in memory.
- Compilers also give us type-safety: We can't operate on two variables whose types are such that the operator doesn't carry any meaning.
- Data Abstractions are of three types: Input, Output and Intermediate
- eg. The output maybe an array, when the input was a matrix and in the process of moving from one to the other it may have went through Stacks, Queues etc.
- Abstracting different notions as data types helps in Re-usability.

5 Lecture 5: Abstract Data Types (Continued!)

- Once, you're familiar with the applications in a domain, its always easier to find abstractions with which you could program those applications.
- If you go from domain to domain, the terminologies they use change, and thus so does the abstractions. Eg. Hospitals and Schools have different abstractions that they care about!
- Language gives you, the very basic abstractions in form of various data types: int(s), float(s), function calls etc.
- Learning Data Structures brings a lot of more ideas about implementing these data abstractions in programs.
- Class represents the abstraction of data. Object is an instance of that class.
- **Functional Representation:** Manipulating these data-abstractions similar to the way we define mathematical equations.
  - Analyzing various aspects of the Data Abstractions, to get an idea of its properties, and then deducing mathematical relations from them, which could simplify our work!
  - The Example of a stack:
    - For a stack, we could easily write that if the stack is `X`, then if the last manipulation of the stack using `push` or `pop` is `push(X,a)`, then `top(X)` will give `a`.
    - Similarly, we also observe that in case, if the last manipulation was `pop`, then what we do is simply go over the latest manipulations and the very time, where we see that the number of usages of the `push` is more than `pop`, than the element which is causing this difference of one would give us the result of `top`.
    - If we were to do this abstraction without using the stack data-structure, then computing the result of `top`, could be calculated in $O(\log n)$ time, where n is the number of such commands we've come across so far.
    - But, that won't be required if we use the stack data-structure(abstraction)!, and then we would be able to perform `push, pop` or `top` in constant time, demonstrating the usefulness of data abstractions.
    - Other Observations would be `empty(new())` will always return true, `removetop(push(X,t))` will give back `X`(i.e. The Stack before `t` was added will be the same as the one we'll get if we remove the top element of the stack after adding the top-element), `not empty(push(X,t))` will be always true as if we push something into the stack than it couldn't possibly be empty.
  - In the last expression, we observe how mathematics could help us better represent the functional aspect of various Data Abstractions.