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.

6 Lecture 6: Abstract Data Types(Still Continued!)

- We want to package our program into different classes and objects, so that they are abstracted and only interact with the outside world through some interfaces, in the same way that electronics and other stuff interact with the world using buttons, Ports.
- These abstractions essentially define a protocol of interacting with these objects.
- Just like any eletronic product has all of its Components, are made in different places and then assembled, so that each of these componenets have their own abstractions, the same stands true for Programming Abstractions.
- When you're the manufacturer, you'd understand everything that happens inside the blackbox of the abstraction, because you'd want to keep it improving, But if you just want it to develop an application of it, you don't care about that. The same paradigms apply to Abstractions in Programming.
- Abstractions usually have a hierarchy, one abstraction used by another which is yet used by another.

- Concepts in Physics, Chemistry, Biology could be all taken to be abstractions, with some commmon abstractions spanning across all subjects.
- The Nouns/Terminology changing gives us the set of abstractions.
- We come up with abstractions so that at the next level, we can forget how these things work, and not care about them!
- Abstractions could thus be taken as a higher level view of stuff.
- **Designing Tips:**
  - Keep Program Compilable throughout.
  - Don't wait till the end to Compile.
  - Define the Functions outside the main class, In the class Declaration keep only the declarations of this function, implement these later.
  - We need to have Destructor/Constructor implementations, if we've declared it before hand, as when we create that object the Constructor would definitely be called and when the program ends, the Destructor would definitely be called.

7 Lecture 7: Abstract Data Types(still continued!)

- Declarations give us Abstractions, and definitions give us implementations.
- Interface is what is available on the object to operate it.
- This is interface comes from the public functions/variables of the Object's Class.
- Any file with any declarations could be compiled as an object file, by using `g++ -c`.
- The Object file could be used later.
- **Encapsulation:** Package all variables/functions together as a single unit, either as a `struct` or `class`. Also, these shouldn't be able to easily accessible from outside this unit!
- Some variables are simply not meant to be global, like phone numbers in a mobile etc., so we shouldn't keep them global, but specific to various classes. Even better keep them private to whatever extent you can.
- If we don't Encapsulate, We need to deal with all componenets separately and hence, we'll have to pass these instances to global functions in order to manipulate them!
- Structures could help us solve the above problems to some extent, but its Classes which finally allows us to increase security further as well as preventing repetitive passing of these Encapsulated Structures to functions(which requires time as it includes copying the entire variable!).
- Big Classes and Big Functions are to be prevented. All these functions, variables for each class should be at the top, so that it could be comprehended easily, and the interactions should be described there itself in comments so that whoever is reading could understand it there itself.
- That's why the declarations should be separated from the definitions!

8 Lecture 8: Abstractions as State Machines

- When we view Abstractions as State Machines, then manipulations made in the main funciton are like Human guards which have access inside the machine. We should keep at least those things in the main() function.
- Every incoming member function call to an object/Abstractions which serves as State Machine must satisfy its pre-conditions . Eg.Can't pop from empty stack, can't push to full one!

- These pre-conditions should be checked inside the abstractions(object!).
- Transitions between the states of these state machine of ours could be viewed as Call to various member functions which are interacting with each other.
- Practice: Drawing SMs for various Abstractions.
- In Complex SMs, We don't show actions which aren't allowed. Simply write if it isn't allowed then give error
- In the Telephone Program, We should have each of the individual states as a different member function so that, when that function is running, then it could be the indication of being in that state.
- Thus, We should've had separate member functions for Dialing, Connecting, Conversing etc.

I didn't make notes for most of the portions in the second half of the First half-sem as I thought it wasn't helping much!
I did realize this course was a vague one and without summarizing the lectures, it becomes pretty clueless for one to figure what to study before exams. Hence:

9 Lecture on May 3: Assertions

- When we write the proof on paper, it has various steps, we would want to make sure these steps were indeed correct and we make sure of this by using assertions, which inform us if we were correct w.r.t the proof.
- Debugger is a post-development way to do so, Assertion is a way to do so during Runtime itself. We have certain breakpoints/checkpoints in the code, where we print something to check, if what we intend to do was correct.
- Exceptions are also a way to handle errors in runtime itself.
- `assert`, helps us check if a particular condition is satisfied at a particular step.
- `assert` is like a demonstration of correctness.
- `assert` is more about finding bugs and you may not add it in the end of the program!
- `try` and `catch` enable us to have error handling in run-time, we generally remove assertions after we've fixed all the bugs!
- `#define NDEBUG` , the assertions are turned off, thereby providing us with a way to not have assertions messing the way our program runs.

10 Lecture on May 5: Aseertions for Program Correctness

- We use Assertions for Program Correctness and then remove it after we're done, its like After building a building we are removing the structures which were put up initially to make sure the building's structure was intact.
- The Program Correctness part is only upto O(n) for any method.
- Functions which check the correctness will be used inside the assert function itself.
- Also, a custom-made macro is called in the same scope from which its called, any method/variable in the scope of the place from where the macro is called, can be used inside the functions part of the assertion's condition or the Macro itself.