CS 246 Fall 9 — Tutorial 9

November 20, 2019

Summary

1 Exception Handling

To handle errors in C++, we use *exceptions*. Exceptions can be thrown and then caught at another point in the program. Code that might fail should be wrapped in a *try/catch block*, and the program can throw any value with `throw <value>;`

To catch anything thrown, use `catch(...)`. When multiple exceptions can be thrown, multiple catches can be used; an exception will go to the first one it can match, and so more specific exceptions should go first.

If there is no try/catch block to handle an exception, the program will terminate. If there are ever two exceptions active at the same time, the program will terminate.

Question: When could this occur?

2 Factory Method Pattern

Problem: At run-time, we want to create polymorphic objects of some class based on some input.

Solution: Give classes a virtual method ("factory method") that produces the appropriate objects.

Example: A "virtual copy constructor" for an object of a polymorphic class. (A4Q2!)

3 Template Method Pattern

Problem: Subclasses need to modify the behaviour of functions, but not in an arbitrary fashion.

Solution: The base class must provide no public virtual functions, instead relying on private virtual functions used within the public functions. Subclasses may override the private functions, but not the public ones.

Example: You are writing a banking library that clients can use to take payments from customers. Different clients need to be able to modify parts of the library for various reasons (coupon codes, gift cards, etc.) To ensure the clients don't write incorrect code that leads to fraud, the template method pattern should be used.

4 Project Advice

The final project in this course is likely one of the first times you've been asked to completely design a program and do so in a team. The following is advice to help the project go well.

Program Design:

- Single Responsibility Principle:
  - Each class should be responsible for one role. For example: reading input, displaying the game state, storing player information, controlling interactions between players.
  - Part of this is designing good interfaces for classes. You should be able to interact with a class you didn't implement without having to look at the code. Giving methods good names helps a lot.
- This is a course in object oriented programming. The projects have been designed and chosen to encourage you to use material you have learned in the course. Use design patterns and inheritance.

Working in a group:

- Don't be a jerk.
- Communicate and don't be afraid to ask your teammates for advice.
- While your entire group should have an understanding of the control flow of the program, you don't need to know how the entire system works. During demos, we may ask to see some code to ensure it's working correctly. This is why all members must be present.
- Consider using git to share code. We recommend using the UW gitlab which is free and you can make private projects. If you use github, or any other service, make sure your project is private. Some git commands:
  - git add <files>: adds <files> to the list of files to be committed.
  - git commit -m "<message>": save all changes to the files that you have added. "<message>" should be a meaningful message about what you have done.
  - git push: adds all commited changes to the current branch. Others will see these changes.

  - git pull: updates the current branch to its most recent state.
  - git checkout <file>: replaces <file> with the version of <file> which you last committed. This removes any changes you have not committed. Be careful!
  - git checkout -b <branch>: creates <branch> and switches to it.
  - git checkout <branch>: switch to <branch> which already exists.
  - git push origin <branch>: pushes <branch> to the repository for others to use.
  - git branch: lists all branches.
  - git merge <branch>: merges the current branch and <branch> to a single branch.

General Advice:

- Occasionally the specification is vague. If the exact behaviour isn't specified, make a reasonable assumption about how to proceed. And check Piazza.
- Don't focus on bonus features until the project is nearly done. The mandatory requirements are worth 60% of the project grade, while bonus is at most an extra 10% and we choose the mark for any bonus feature based on difficulty of the feature. That being said, when designing your project, consider what bonus features you may want to implement as designing the project to be extendable will make adding bonus features more simple if you have time.
- Compile and test your code frequently. If possible, test features as you implement them. It's very useful to implement a display first and slowly add functionality and test as you go. Saving all compiling and testing for the end is a bad idea.
- If you plan to use smart pointers for heap allocation, do so from the beginning. Don't try to do it as a last minute thought.
- Start early and give yourself more time than you think things will take.
- Choose the project that looks most interesting. You'll do better if you are interested in what you are doing.