CS 246 Fall 2018 — Tutorial 9

November 14, 2018

Summary

1 Exception Handling

- Instead of using C-style strategies of handling errors, we use **exceptions** in C++ to control the behaviour of the program when errors arise.
- From a previous tutorial, if an exception is *raised/thrown*, the program will terminate if there is no handler (that is, a `catch` block) for it.
- Recall that we catch exceptions with try and catch blocks:

```
try {
    throw 42;
}
catch (...) { // "..." will accept any type of error
    cerr << "Caught something" << endl;
}
```

  The `try` block will run as normal, but any errors will be handled by the `catch` block underneath.
- **Note:** if the `catch(...)` is used, it must be the very last catch clause.
- The `try` and `catch` blocks must be used together.
- We can throw *anything*, including exception objects, strings, integers, etc.
- We can also `throw` what we just caught in the `catch` block:

```
try {
     throw 42;
}
catch (...) {
    throw; // will throw the same exception that was caught
}
```

1.1 Exception and Inheritance

- Consider the code below:

```
class BadInput {};

class BadNumber : public BadInput {
public:
    string what() {
        return "no number given";
    }
};

int main() {
    try {
        int x;
        cin >> x;
        if (x < 50) {
            throw BadNumber{};
        }
    }
    catch (BadInput &b) {
        cerr << "oops" << endl;
    }
    catch (BadNumber &b) {
        // accessing auxilliary information from object that was caught
        cerr << b.what() << endl;
    }
}
```

  What will be the output?

- Since `BadNumber` inherits from `BadInput`, it will be caught by the first error handler.
- **Good practice:**
  - Throw by value, and catch by reference.
  - `catch` blocks needs to be ordered from the most specific to the least specific.
    - For example, if there is an inheritance relationship between the exception classes, you should always catch the subclass exceptions first.
    - `catch(...)` must always be the last catch statement.

2 Factory Method Pattern

- **Problem:** At run-time, we want to choose a subclass of some superclass and create objects of that class.
- **Solution:** Create an abstract class that has a method that creates an object of the superclass. For each subclass, create another class that inherits from the abstract factory class and implement the method to create objects of that subclass.
- **Example:** Vending machine (`09/factory_method`)

3 Template Method Pattern

- **Problem:** We want to allow subclasses to have different implementations for some sections of a method, but enforcing a structure of the method and not allow subclasses to have different implementations for other sections.
- **Solution:** Implement an abstract superclass which has public non-virtual method(s) and protected/private virtual method(s).
  - Have the superclass non-virtual method(s) perform the operations which will be the same for all subclasses.
  - The subclasses can implement the virtual portion(s) in a way that suits their needs.
  - If the virtual method is protected, then in our override, we can call the superclass version and add additional code.
  - If the virtual method is private, we cannot call the superclass version, but we can still override the method.
- **Example:** Faces (`09/template_method/face`) and turtles (`09/template_method/turtle`).

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.