C++ Project, Baruch C++ Course

Originator: Daniel J. Duffy, Datasim

Summary and Goals

The objective of this document is to develop ideas to help you *define and execute a software project.* The process in the document is generic and can be used to design a software system as a set of loosely-coupled and independent subsystems having well-defined and flexible interfaces.

The methods and techniques in this document can be applied to many kinds of software projects. In other words, it is a defined process that you can and should customise to the application in question.

This document should be used as a kind of template or roadmap to help you keep focused on designing maintainable software systems using C++11.

Background

The Baruch C++ course contains six major modules:

1. Essential C++11/C++14 syntax and language features.
2. Advanced C++ syntax.
3. Multithreading, parallel programming and STL algorithms.
4. Boost and STL libraries.
5. Object-oriented design patterns.
6. Next-generation design patterns and Unified Software Design (USD).

The first three modules focus on C++11 syntax, language features that extend and improve the C++ language. We recommend that you use these features in your application code rather than C++98 syntax whenever possible. We also discussed C++ libraries in modules 3 and 4 and their contain ADTs and algorithms that you can use rather than creating your own home-grown version. We also discussed the *functional programming model* that complements the *object-oriented programming model* and in some cases is an improvement on the latter approach.

Levels 5 and 6 concentrate on the famous GOF design patterns and a defined process to analyse and design complex software systems, respectively. These levels build on the language features and libraries of the first four chapters. These levels constitute the preparations before embarking on the final project. In general, we use C++11 to help us design maintainable, readable and standardised software systems. To this end, this is my top ten list of useful features (in no particular order):

1. Lambda functions and captured variables in combination with creational design patterns to configure applications.
2. Variadic (and non-variadic) tuples to collect and organise heterogeneous data types.
3. Universal function wrappers (`std::function<>`) as an alternative to subtype polymorphism and class hierarchies.
4. Use smart pointer rather than raw pointers in general.
5. Use STL data structures and algorithms rather than home-grown code.
6. Use Boost C++ libraries rather than home-grown code.
7. An optimization step is to look for places in your code that can be parallelized using threads and tasks in *C++ Concurrency*. Can you draw a task dependency graph for your problem?
8. Use C++11 and Boost C++ libraries for random number generation and statistical distributions.
9. Consider using Boost *signals2* to model events and callback functions between loosely-coupled software components.
10. Use external matrix and regular expression libraries. Don't reinvent the wheel.

You may have your own preferences in addition to this list. We recommend that you do a quick review of these topics in order to refresh your knowledge before embarking on the project.

Problem Description and Preparation

The main goal of the student project is to show how to apply the material of Levels 1 to 6 to create a small software system that is maintainable, understandable (for example, well-documented code) and that satisfies the requirements of the problem specifications.

Since the project has a limited duration (typically, two weeks) it is important that you define scope of the project in such a way that you reach your objectives within that period of time. The focus is on producing accurate output for a range of input parameters. If there is time left you can focus on other issues such as input validation, exception handling and code refactoring (if applicable).

Some tips and guidelines:

a. Draw up a *feature list* that describes the functionality that the software system should satisfy.
b. Assemble background information, documentation and articles that describe the problem.
c. Carry out project estimation and planning; break the project into a series of software prototypes in increasing levels of functionality.
d. Get it working, then get it right, then (and only then) get it optimised.

Once you understand the problem then you start to design it.

Software Process

This is described in detail in references/chapters A and B below. A summary of the design process is:

- Phase I: System Scoping and Initial Decomposition.
- Phases II: System Components and Interfaces.
- Phase III: Detailed Design.
- Phase IV: Implementation, Review and Maintenance.

In the context of C++ these phases can be executed as follows:

1. *Align* the feature list with the systems in the system context diagram; which features are the responsibility of which system. Ideally, a feature should be implemented a single system.
2. Check: all features in the feature list are taken care of by the systems in the context diagram.
3. The responsibilities of each system must be determined (*provides/requires services*).
4. Determine and implement inter-system interfaces (using subtype polymorphism or universal function wrappers, for example).
5. Design and implement each subsystem in C++.
6. Test, debug and generalise the code for each system.

We note that these steps are executed iteratively and backtracking is possible or even necessary as we discover that we have forgotten to implement a given feature or that the design needs to be generalised and improved as we embark on the next software prototype.

Structure of Project and Attention Points

We suggest a C++ project based on the requirements and solution in reference A below. We suggest the series of prototypes:
P1: Implement the Context diagram for one-factor plain options (GBM) using the Euler method and Mersenne Twister.
P2: P1 + Support for other kinds of options (e.g. CEV, CIR, barrier, Asian) using Euler method.
P3: P2 + support for other FD schemes (e.g. Milstein, drift-adjusted predictor-corrector).
P4: Creational patterns to construct all the components in prototype P3 using a combination of *Builder* and `std::tuple`.

Create a separate directory for each prototype so that each one can be run and tested separately.

Input: test the software for a range of put and call options for a range of option data parameters. Report on and analyse your results.

NOTE

Run the project in Release mode (Debug mode is ~ 10 times slower).
Run the project in Release mode (Debug mode is ~ 10 times slower).
Run the project in Release mode (Debug mode is ~ 10 times slower).

Background Reading

https://en.wikipedia.org/wiki/How_to_Solve_It

A: And the chapters on design applications to Monte Carlo (C# treatment; the C+ + design will rather similar). Officially “Application Design in C#: The Monte Carlo Method for Option Pricing”.
B: Chapter 9 “An Introduction to Unified Software Design (USD)”.

Other Future Projects based on the defined Process

TBD for September 2016 cohort.