ECE 449/590 – OOP and Machine Learning
Lecture 06 Managing C++ Projects

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 12, 2022

Outline

Software Engineering

C++ Building Process

Header Files and Translation Units

Reading Assignment

- This lecture: Project 2 Instruction
- Next lecture: Accelerated C++ 10, 5

Outline

Software Engineering

C++ Building Process

Header Files and Translation Units

Software

- What is the difference between software and programs written for course projects?
- More features? Better quality? Higher performance?
- Software is designed in a way so it can be improved and reused.
- Software engineering practices attempt to define a process to reduce the overall risk.

The Waterfall Model

- A conventional process of software development.

Stage 1. Requirements Analysis and Definition
Stage 2. System and Software Design
Stage 3. Programming and Unit Testing
Stage 4. Integration and System Testing
Stage 5. Operation and Maintenance

- Waterfall: never go back and revise previous stages
- Advantages
  - Detailed planning for time/personnel/budget within the constraints
  - Goals are well-defined within each stage

The Waterfall Model: Challenges Nowadays

- Requirements analysis and definition assumes that clients know what they want.
  - Not quite true as clients usually learn how software can actually help them during the development process.
- System and software design assumes that some expert can make a feasible plan.
  - There may not be an expert if you are building something new or if you cannot afford one.
- Integration and system testing assumes the developers can resolve issues quickly among themselves.
  - Communication is required but expensive, especially when people leave or enter the team.
- Demo is only possible after integration and system testing.
  - Too late and very risky if delay happens.
- Operation and maintenance could be difficult as bugs may come from OS and supporting libraries.
  - Upgrading may break the whole system.
- Overall not flexible for the rapidly changing world nowadays.

Agile Software Development

- A set of software development methods that teams may choose to satisfy their needs for a specific project.
- Our choices:
  - Iterative and incremental development (IID)
  - Test-driven development (TDD)
  - Code refactoring
  - Continuous integration (CI)

Iterative and Incremental Development (IID)

- An incremental/iterative cycle: build a small portion or make a small revision.
  - The whole system can be assembled and improved across multiple cycles.
  - Progress can be demonstrated at the end of each cycle.
  - Utilize new understandings learned from previous cycles in the next cycle.
- Allow us to make progress without knowing much of machine learning, EasyNN, Python, and C++.
  - So we will have the whole semester to work on EasyNN instead of a few weeks.
- Within each incremental/iterative cycle, the waterfall model can be applied.
  - Changes are limited – much less risky than applying it to the whole system

Test-Driven Development (TDD)

- Unit testing: for a small unit, e.g. a class
  - Serve as an executable specification of your code, more precise and consistent than documents in plain English.
- Integration testing: for the whole system
  - A good chance to demo to clients and to communicate with them to understand their requirements.
- Acceptance testing: determine whether requirements are met.
- Test-driven development
  - Tests are created before writing the code.
  - The whole system is decomposed into testable pieces.
- We grade your implementation by acceptance testing.
  - The scope of many of our tests is similar to unit tests in realistic industrial settings.

Code Refactoring

- How to modify code to accommodate more functionalities without breaking existing ones?
  - Assume old and new tests are available as TDD requires.
- Two steps of code refactoring
  - Modify code without adding more functionalities and validate your changes using old tests.
  - Add more functionalities and validate with new tests.
- What to refactor?
  - Use a function to break a long program into smaller pieces.
  - Use a class type to organize data.
  - The purpose of refactoring is to make it easier to change your code.

Continuous Integration (CI)

- Manual unit testing is tedious.
  - Need to run many tests and wait.
- Manual integration testing is difficult.
  - Especially if the team is big or the production environment differs from the development environment.
- However, frequent testings are always desired to guarantee good quality.
- Continuous integration: enable automated and frequent testing in production environment.
  - As frequent as anyone like while requiring little to no effort from developers. The only constraint is the available computational resource for testing.
  - A CI system usually depends on a revision control system, which provides additional benefits of storing all versions of your source code.
- We use Git as the revision control system and utilize a CI system to test your projects against grading test cases.

Projects Outline

- Project 2: ctypes and Scalar Operations
  - Python and C++ interoperation.
  - Evaluating expressions with scalar operations.
- Project 3: Matrix Operations
  - Python and C++ interoperation with NumPy arrays.
  - C++ class design to support matrices and tensors.
- Project 4: NN Inference
  - Implement tensor operators for neural networks.
- Project 5: MLP Training
  - Implement back-propagation on fully connected layers.
- Project 6: CNN Training
  - Implement back-propagation on convolutional layers.

Outline

Software Engineering

C++ Building Process

Header Files and Translation Units

Building

- Building: create an executable from a set of source files
- What is the building process for C++ programs?
- Why is it designed that way?
- How should we leverage it to improve productivity?
  - Reuse existing codes/libraries – save coding time!
  - Catch typos and mistakes – save debugging time!

Compiler and Compiling

- Compiling is the process to turn source codes into binary forms.
- Many typos and mistakes can be identified by compiler during compiling.
  - The compiler will refuse to generate binary codes until you correct all compiling errors.
- Example 1: undeclared names
  - Is there a typo in your variable or function name?
  - Are you trying to access a variable or function out of its scope?
- Example 2: unmatched types
  - Are you assigning a wrong value to a variable?
  - Are the arguments to a function passed in wrong order?

Correcting Compiling Errors

- Don't get frustrated if your code won't compile successfully.
- Always start with the first error message – the remaining ones may simply go away if the first one is corrected.
  - Repeat when necessary until all errors are corrected
- Read the error message carefully to determine what's wrong.
  - An error message usually starts with the file name and line # that you can locate the line causing the error – IDEs may even bring you to the line directly if you double click the error message.
  - Try to follow the description of the error thereafter
  - Only a few types of error messages appear frequently and you will quickly get familiar with them as you code your projects.

Separate Compilation

- If we put everything into a single C++ file, even with functions and data types, issues arise when the program becomes more complicated.
  - There are chances some code is changed accidentally.
  - It takes longer to compile a larger file.
- Separate compilation: use multiple files for a program
  - Reduce complexity by separating interfaces and implementations
  - Reduce build time by compiling modified files only
  - Reduce chances of accidental changes by copying files instead of code snippets

Building, Compiling, and Linking

- Separate compilation consists of two stages.
  - Usually the whole process is known as building.
- Stage 1: compiling (by compilers)
  - Convert each source file into an object file (the word "object" here has nothing to do with OOP)
  - Source files may be written in different programming languages as long as the interfaces are compatible.
- Stage 2: linking (by a linker)
  - Combine object files into one executable
  - Pre-compiled object files can be provided by a third-party who otherwise won't provide source files.
  - Pre-compiled object files are usually organized into library files.
  - Object files and libraries can be linked at runtime dynamically to extend functionality of existing programs, e.g. for our course projects.

Build Management

- Compiling
  - For each source file, determine the compiler to compile it
  - To save compiling time, only modified source files should be compiled for each successive build.
  - Determine compiling options
- Linking
  - Locate the necessary library files
  - Determine the linking options
- A build management tool is used to manage all such details.
  - E.g. GNU Make for our projects.

Outline

Software Engineering

C++ Building Process

Header Files and Translation Units

Function Interface and Implementation

```
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ... // implementation omitted
}
```

- Interface: inputs and outputs of the function, i.e. parameter and return types
- Implementation: the method to generate outputs from given inputs, i.e. function body
- Separate compilation for C++
  - The compiler only need to know the interfaces
  - The linker will locate the implementations

Function Declaration

```
// libeasynn.h
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs);
```

- Function declaration
  - Define the interface to the function for the compiler
  - Function header followed by ;

Function Declaration (Cont.)

```
// libeasynn.h
extern "C"
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs);
```

- Function declaration may include additional properties for the functions.
    - E.g. `extern "C"`.
- `extern "C"` makes a C++ function to have 'C' linkage.
    - For other languages, functions with 'C' linkage are simpler and easier to work with than usual C++ functions.
    - However, functions with 'C' linkage are more error-prone so we only use them when absolutely necessary.

Function Definition

```
// libeasynn.cpp
void append_expression(program *prog, int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ...
}
```

- Function definition
  - Function header + function body
  - Define the implementation of the function
- There should be exactly one definition for each function among all the source files.

Header Files

- Unlike most modern languages where compilers utilize certain form of “import” to manage declarations and definitions, C++ inherits from C to NOT manage them by compilers.
  - Each source file should contain the declarations for the functions it intends to use, plus user-defined data types used in those declarations.
- Follow the solution from C to organize declarations and class types into header files and to allow source files to access them.
  - Help to maintain consistency among multiple source files.
  - No mismatch of function declarations in multiple source files.
  - No mismatch of class types in multiple source files.
  - Mismatched function declaration and definition will be caught by linker.

Translation Unit

```
// libeasynn.cpp
#include <stdio.h>
#include "libeasynn.h"
#include "program.h"
#include "evaluation.h"

program *create_program() {
    ...
}

...
```

- Use `#include` with `""` to include your header files
  - You can include header files from other header files.
- The source file, together with all the header files it includes directly or indirectly (through other header files), is called a translation unit.

#include Guard

```
#ifndef PROGRAM_H
#define PROGRAM_H

class program {
}; // class program

#endif
```

- A header file may appear more than once in a translation unit, resulting in compiling errors, e.g.,
  - A data type from the header file is defined more than once.
  - There may exist cyclic dependences among header files.
- Use a `#include` guard to prevent a header to appear more than once in a translation unit.
  - The header is still included multiple times, though it is only seen by the compiler for the first time.
- Don't forget the `;` at the end of the class definition.
  - If you see strange compiling error messages, double check all your class definitions for the `;`.

Working with Classes

```
// program.cpp
#include "program.h"
#include "evaluation.h"

program::program() {
    ...
}

void program::append_expression(int expr_id,
    const char *op_name, const char *op_type,
    int inputs[], int num_inputs) {
    ...
}

int program::add_op_param_double(
    const char *key, double value) {
    ...
}

evaluation *program::build() {
    ...
}
```

Managing Your C++ Files for Course Projects

- All your source files (.cpp) and header files (.h) should be stored in `src`.
  - Otherwise, the compiler may fail to find some of your files when building your project.
  - Unused .cpp and .h files should be removed.
- `easynn_test.cpp` is provided to help you debugging your C++ implementaions.
  - It should be stored outside `src` as it is not part of your C++ implementation.
  - Don't modify it until it works properly!

Resolving Common Linking Errors

- You may receive linking errors when running Python code for our projects as linking happens when loading a shared library.
- Multiple definitions of a function: there should be exactly one definition for each function among all the source files.
  - Did you put a function definition instead of its declaration into a header file?
- Missing definition of a function.
  - Make sure the parameters of the function declaration matches its definition.
  - Make sure you have 'git add' the .cpp file containing the function definition.
- Multiple definitions of a class/struct.
  - Don't put a class/struct definition into a .cpp file.
  - Did you use #include guard for all your header files?
- If you still have troubles, talk to us immediately.
  - Don't waste you time – it may take you days to figure out some trivial mistakes when setting up separate compilation.

Summary

- Software engineering practices: Waterfall vs. Agile
- Separate compilation supports the separation of interfaces and implementations.
  - In a translation unit, functions should be declared and types should be defined before being used.
  - Use header files to manage function declarations and type definitions
  - Use `#include` guard in every header file