Lab 2

Table of Contents

Intro to Makefile

Code Compilation

- Suppose we have three files:
  - main.cpp, factorial.cpp, printhello.cpp
- To compile the project, we type:
  - g++ main.cpp factorial.cpp printhello.cpp -o main
- In summary, suppose we only have one main, we type:
  - g++ <all .cpp files> -o <executable name>

Linking

- Suppose there are 10,000 .cpp files
  - Compiling in the way above is going to take a long time
  - Solution: we compile one by one (i.e. generate .o files)
- General format:
  - g++ <filename>.cpp -c
    - will generate <filename>.o
  - Ex:
    - g++ main.cpp -c
    - g++ printhello.cpp -c
    - g++ factorial.cpp -c
    - g++ main.o printhello.o factorial.o
- Advantage: we don’t need to recompile unmodified files in this way.

Makefile

- Still, typing g++ <cpp filename> -c by hand one by one is going to take a lot of time...
- So we use Makefile to automate this process!

Makefile, Version 1

```
1  ## VERSION 1
2
3  hello: main.cpp factorial.cpp printhello.cpp
4      g++ -o hello main.cpp factorial.cpp printhello.cpp
```

- Line 3:
  - hello is the **target**. Its generation depends on the **objects** on the right hand side.
- Line 4:
  - This tells the terminal how to generate hello, the target above
  - Tab is necessary, not spaces.
- To run Makefile, type *make*.
  - If your makefile is not named Makefile, type:
    - make -f <makefile name>
- make will also check if files are modified

Makefile, Version 2

- Version 1 is bad because it takes a long time to compile 10,000 files
- Line 7-9
  - Define cxx, target, and obj
- Line 11
  - Target is dependent on obj
- Line 12
  - Rule for generating target
- Line 14-21
  - Tells how to generate the object files

```
6   ## VERSION 2
7   CXX = g++
8   TARGET = hello
9   OBJ = main.o printhello.o factorial.o
10
11  $(TARGET): $(OBJ)
12      $(CXX) -o $(TARGET) $(OBJ)
13
14  main.o: main.cpp
15      $(CXX) -c main.cpp
16
17  printhello.o: printhello.cpp
18      $(CXX) -c printhello.cpp
19
20  factorial.o: factorial.cpp
21      $(CXX) -c factorial.cpp
```

Makefile, Version 3

```
23  ## VERSION 3
24  CXX = g++
25  TARGET = hello
26  OBJ = main.o printhello.o factorial.o
27  CXXFLAGS = -c -Wall
28
29  $(TARGET): $(OBJ)
30      $(CXX) -o $@ $^
31
32  %.o: %.cpp
33      $(CXX) $(CXXFLAGS) $< -o $@
34
35  .PHONY: clean
36  clean:
37      rm -f *.o $(TARGET)
```

- Cleaner!
- Line 26
  - When adding new files, just add <filename>.o here
  - Nowhere else needs to change
- Line 27
  - More flags
- Line 30
  - $@ means target
  - $^ means all dependencies above
- Line 32
  - All .o files depends on the cpp file w/ the same name
- Line 33
  - $< means the first dependent file
- Line 35
  - Avoid extreme case when there is a file named clean i.e. *make clean* may not work
- Line 36-37
  - After typing *make clean*, remove all .o files and target file

Makefile, Version 4

- Everything is automated
  - Even if new files are added, we don’t need to change our makefile.
- Line 42
  - src is all the .cpp files under current directory
- Line 43
  - obj is all the .o files with the same filename in our src.
- You are not expected to write a makefile like this
  - Version 2 and 3 are good enough

```
39  ## VERSION 4
40  CXX = g++
41  TARGET = hello
42  SRC = $(wildcard *.cpp)
43  OBJ = $(patsubst %.cpp, %.o, $(SRC))
44
45  CXXFLAGS = -c -Wall
46
47  $(TARGET): $(OBJ)
48      $(CXX) -o $@ $^
49
50  %.o: %.cpp
51      $(CXX) $(CXXFLAGS) $< -o $@
52
53  .PHONY: clean
54  clean:
55      rm -f *.o $(TARGET)
```

Lab 2 Description

Project Structure

- README.txt
- Card
  - card.h
  - card.cxx
  - cardmain.cxx
- Deck
  - deck.h
  - deck.cxx
  - shuffle.cxx
  - deckmain.cxx
- Poker
  - poker.h
  - poker.cxx
  - pokermain.cxx
- Makefile

Card (Class Initialization and Constructor)

- Represents a card object
- Card(suit_t suit = 0, rank_t rank = 0)
  - suit and rank are integer type
  - Suit ranges from 1-4 (club, diamond, heart, spade), rank ranges from 2-14 (2-10, J, Q, K, A)
    - Do not worry about letting ‘A’ be both 1 and 14 for now
- See the rest of the requirements in lab document

Card (Operator Overloading)

- friend std::ostream& operator<<(std::ostream& os, const Card &c);
  - prints the suit followed by the rank of the card
  - why friend keyword?
- operator int() const
  - returns a numeric order for the card
- Resources
  - https://stackoverflow.com/questions/15999123/const-before-parameter-vs-const-after-function-name-in-c

Deck (Class Initialization and Constructor)

- Represents a deck of cards
  - Optional: have multiple decks for the deck class
- Private Variables
  - int nCards // optional, number of cards total
  - Card cards[CARDS_PER_DECK] // a deck of cards
  - int next // index of next card to be drawn
  - int guard // threshold value for shuffling
  - ... // more?
- Functions to be implemented
  - Card& deal()
  - ostream operator<<(ostream& os, const Deck& d)
  - void shuffle() // implement this in shuffle.cxx
  - ... // more?

Deck (Operator Overloading)

- friend std::ostream& operator<<(std::ostream& os, const Deck &d);
  - prints all the 52 cards in the deck, 13 in a row.

Deck (Shuffle)

- void Deck::shuffle(void)
  - implement in shuffle.cxx
- Fisher-Yates Shuffle (dirty) **Recommended**

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n - 1 downto 1 do
       j = random integer with 0 <= j <= i
       exchange a[j] and a[i]
```

- Fisher-Yates Shuffle (clean)
  - make a shuffled copy of the original deck
  - use more memory as need to make a copy
  - slower also because need to make a copy

Poker

- A hand consists of 5 cards. They are ranked in the descending order below:
  - Straight Flush ➔ Quad(4 of a kind) ➔ Full House ➔ Flush ➔ Stright ➔ Triple ➔ 2Pair ➔ Pair ➔ High
- Task 1: Given a hand, obtain what type of hand it is.
  - Check CodingPokerRanks.pdf when you work on it
- Task 2: Collect probabilities of drawing each type of hand
  - deal() sufficient amount of hands, print the result.

Poker (Operator Overloading)

- ostream& operator<<(ostream& os, const Poker &h)
  - print the hand of 5 cards and the name of their pattern (flush, straight, ...)

Object-Oriented Programming (OOP)

Objected-Oriented Programming

- There are 4 principles for object oriented programming
  - Encapsulation
    - **Data hiding** from the outside world
      - E.g. Set variables private or local
      - E.g. Setters and getters
  - Inheritance
    - Not needed for lab 2 – might go over in the future
  - Polymorphism
    - Not needed for lab 2 – might go over in the future
  - Abstraction
    - Hide unnecessary details. Gain information and/or solve problems **at interface level**.
      - E.g. Separation of .h and .cpp
      - E.g. Write *private* helper functions to solve more complex problems

Encapsulation & abstraction, more examples

- Bad:

```
if (/*rule 1*/ && /*rule 2*/ && /*...*/) return 6;
```

- Good:

```
if (isFullHouse()) return POKER_FULLHOUSE;
```

  - Create helper functions, if needed
  - No hard-coded values

Encapsulation & abstraction, more examples

- Your main function should look clean overall
- Not ideal:

```
int main(void) {
    Poker poker;
    cout << "Sample hand for each Rank:" << endl;
    /**
    50 lines of implementations
    */
    cout << endl << "Statistics:" << endl;
    /**
    Another 50 lines of implementations
    */
    return EXIT_SUCCESS;
}
```

- Ideal:

```
int main(void) {
    Poker poker;
    cout << "Sample hand for each Rank:" << endl;
    pokerHands(poker);
    cout << endl << "Statistics:" << endl;
    pokerStats(poker);

    return EXIT_SUCCESS;
}
```

Member Functions vs. Non-member Functions

- Member functions are within the class scope, whereas non-members are outside of class scope
- When deciding whether a function is a member function, make sure it makes sense (literally) to be put inside the class.
  - Suppose I have a class named Node
    - Functions like setValue(), next(), operator+() can be member functions
    - Functions like eat(), copyList(), operator <<() should not be member functions
      - eat() is irrelevant to Node
      - copyList() is more like an attribute for a list of nodes, not node itself
      - operator <<() has to be a member of the ostream class
- Non-member functions cannot access private members

Friend Functions

- Non-member functions with a friend keyword
- It allows non-member functions to access a class's private variables.
  - e.g. friend ostream operator << (ostream os, const Deck& d);
  - e.g. friend sequence& operator + (const sequence& s1, const sequence& s2);
- Use *friend* keyword for a non-member function if and only if you have to access private variables to implement this function.

Public Functions vs. Private Functions

- A function is private when users don’t need to see the function
  - Make your API look compact and clean
  - When future programmers use your code, they can complete their task without using these private functions
  - E.g. *isFlush(), isStraight(),...* are private
    - rankHand() helps you to rank your hand already.

Decision Tree

Given a function:

- if (isRelatedToClass()) // member function
  - if (isNeededByUser())
    - return PUBLIC
  - else
    - return PRIVATE
- else // non-member function
  - if (hasToAccessPrivateVars())
    - return FRIEND
  - else
    - return NON_MEMBER
- There are a few exceptions in C++, but we don’t worry about it for now.

Advanced Programming Tips

Bitmask

- We can view an integer as a binary number
- Bitwise Operators include &, |, ^, etc.
- Bitmasking can help to remove “unnecessary information” of a number
  - e.g. $(1011\ 0111)_2$ & 0x1F == $(1011\ 0111)_2$ & $(0001\ 1111)_2$ == $(0001\ 0111)_2$
- Bit Shift operators >>, <<
  - E.g. $(0011\ 0111)_2$ >> 2 ==$(0000\ 1101)_2$, $(0011\ 0111)_2$ << 2 == $(1101\ 1100)_2$

Monte Carlo Random Sampling

- Use randomized algorithm to obtain satisfied result
- E.g. Approximate the value of π
  - How: Note when radius is 1, the area is equal to π
  - So we can put a lot of random points inside a 1*1 square, and calculate the percentage of the number of points inside ¼ circle of a 1*1 square!
    - Note $r_i=\sqrt{(x_i^2+y_i^2)}$. Therefore, when $r_i <= 1$, the point is inside the circle.
  - π = 4*(points_in_quarter_circle/ total_num_points)

Threshold Value ε

- It can be used to
  - Set a stopping criterion of a while loop (what you might apply for this lab)
    - e.g. Check convergence of series
  - Check equivalence of two real numbers
    - Due to rounding errors, sometimes two doubles are mathematically the same, but they differ slightly in the program.
    - Solution: set ε to be a very small value. If (abs(a-b) < ε), then a==b.
    - Intuition: (see below)

Definition 0.1. We say two real numbers a and b are **equal** if

$$\forall \epsilon > 0, |a - b| < \epsilon$$

Deliverables

- All .cpp files, including three mains
- All .h files
- Makefile

Demo

- Code compilation/run

Other Tips

- Test code frequently
- Test your code comprehensively
  - Think *carefully* about what needs to be tested
  - Points will be deducted if you missed critical test cases
- ~~You are expected to spend extra time outside of class to complete this lab~~
  - You may or may not need extra time, but feel free to let us know how much time you spend on this one

Don’t Forget

- Submit & demo the code before next week before the lab starts
- File with guide to implement and hints are in Camino
  - Make sure your code can run on school Linux server