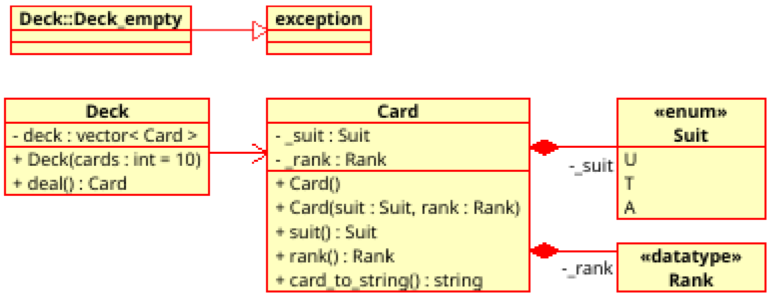
Playing Well
Deck of Cards

CSE 1325 – Fall 2018 – Homework #3
Due Tuesday, September 11 at 8:00 am

Assignment Overview

Having written a basic class in C++, it's time to spread your wings into multiple classes – a Deck of Cards with 3 suits (U, T, and A), and ranks from 0 to 9. We'll also practice some basics: Makefiles, default parameters, exceptions, regression tests, and (at the bonus level) separate compilation.

Full Credit: Using git (with *at least* 3 commits) and a Makefile:



- **Write the Card class in file card.cpp** as shown to the right, with a **regression test in file test_card.cpp.**[1] If the Card constructor's Suit or Rank parameter is invalid, throw a runtime_error.
- **Write the Deck class in file deck.cpp** as shown to the right, with a **regression test in file test_deck.cpp.** If Deck::deal is called when the deck is empty, throw a Deck_empty exception.
- **Write an application in file main.cpp** that instances and prints out a deck of 20 random cards (you do NOT need to ensure that each card is unique!).

Deliver file CSE1325_03.zip to Blackboard. In **subdirectory "full_credit"**, include your full git repository, including your most recent files **card.cpp, deck.cpp, main.cpp, Makefile,** and screenshots **main.png** (your regression tests and main), and **git.png** (git log).

Bonus: Using git (3+ additional commits) and a copy of the above code, **refactor your classes to use separate compilation** by splitting the code into a .h (interface) file and a .cpp (implementation) file. Update your Makefile, regression tests, and application to support separate compilation.

In CSE1325_03.zip **subdirectory "bonus"**, include your git repository with updated versions of **all source files** plus updated screenshots **main.png** and **git.png**.

Extreme Bonus: Using git and the Bonus code, extend Deck to act like a "real" deck of cards. Populate with 30 unique "shuffled" cards by default (using a new public "shuffle" method), plus other enhancements you need. Write a simple game (your choice) using the enhanced Deck. Update the UML class diagram and tests as needed.

---

1 Regression tests have their own main() function, and are built separately from the application. Think multiple Makefile targets.

Notes

Use Git

You are REQUIRED to use git locally (not GitHub, unless you so choose[2]) while developing this code. You will be required to show a screenshot of your git commit log at each level as part of the archive you submit to Blackboard, and will lose points if you don't. **This is not hard! Check the Notes in Homework #2 for help if you can't remember.**

Use a Makefile

You are REQUIRED to use a simple Makefile that can build and execute your main program given the simple command "**make**" (or Tools → External Tools → Build in Gedit), and also support "make test" to build and run all of your regression tests. Here's a working example to get you started (this is the last homework with a suggested Makefile!). Remember: the indentations are tabs, not spaces!

```
# Suggested Makefile for CSE_1325 Homework #3 full_credit
CXXFLAGS += --std=c++17

# Build the main application on a "make" command
main: main.cpp card.cpp deck.cpp
	${CXX} ${CXXFLAGS} -o main main.cpp

# Build and run both regression tests on a "make test" command
test: test_deck test_card
	@./test_card
	@./test_deck
test_deck: test_deck.cpp deck.cpp card.cpp
	${CXX} ${CXXFLAGS} -o test_deck test_deck.cpp
test_card: test_card.cpp card.cpp
	${CXX} ${CXXFLAGS} -o test_card test_card.cpp
clean:
	-rm -f *.o a.out test_card test_deck main
```

Full Credit

Writing the Rank and Suit types

Your first decision is how to represent the Suit (U, T, or A) and Rank (0 through 9) for a card. For this assignment, **use an enum for Suit and an int** (with *optional* typedef from Lecture 05) **for Rank**.

You will also need to determine how to represent the number of suits (since enums are not classes and thus have no .size() method) and number of ranks (or min and max ranks) for your program.

Finally, you will (eventually) need a way to convert the suit back into a char ('U', 'T', or 'A') - this may be via a function[3], a method in Card, a vector of strings subscripted by the enumerated suit, a substring, or any other approach you prefer.

2 If you decide to use Github, please include a text file named GitHub.txt with a link to your public GitHub repository in CSE1325_02.zip.

3 I know – NOT object-oriented. But we'll allow a small exception for enum "helper functions" due to the design of C++ enums, which is not your fault.

You may need to switch between variables of type Suit and variables of type int. You may use C casting for now, e.g., **Suit s = A; int i = (int)s;**

These type(s) may be in their own file(s), e.g., rank.cpp and suit.cpp, or in cards.cpp, as you prefer.

Writing the Card class

Write the card class and store it in a file named **card.cpp**. You are not required to use the underscore in front of the private variables.

- You'll need a constructor where the suit and rank are passed as parameters and simply stored in the private variables. If the suit or rank is not valid, throw a runtime error with a text message as a parameter, e.g., "Suit: Out of range". Do NOT print anything – **libraries throw exceptions, they don't print error messages!**

  You'll need a second constructor where the suit and rank are selected at random. You can include the cstdlib library (which is the same as the stdlib.h library from C), in which case the rand() function will return a random positive integer. Thus, e.g., rand()%10 will give you a random integer from 1 to 9. It's not required that you "seed" the random number generator, though you may if you like.

- You'll need two getters, one for suit (suit()) and one for rank (rank()). These just return the respective private variables. Thus, suit and rank are effectively read-only due to our *encapsulation*.
- You'll need a card_to_string() method eventually for the main application. The standard to_string function should work with the rank, but you'll need some way to convert a suit to a 'U', 'T', or 'A' (see above). A card should print as A3, T0, U9, or similar.

Writing the test_card regression test

You'll write a main() function in a file named test_card.cpp that tests the following 4 cases. Remember to generate no output when the test succeeds (except for case #1 – we don't have the string manipulation tools yet to detect failure there, so you'll have to rely on your eyes).

1. Normative case: Create a deck with 20 cards and stream them to cout. Verify that your card_to_string is working, and that all of the generated cards are valid. (In the future, we'll learn how to automate this test!) By the end, you will have moved this test to be your main application, so that your tests will generate no output.
2. Normative case with explicit constructor case: Create card U3, and verify the suit is U and the rank is 3. Stream an error to cerr if not, or if an exception is generated.
3. Invalid suit case: Verify that creating a card with an invalid suit throws a runtime error. You may need to use a simple cast to pass an invalid suit, e.g., (Suit)3.
4. Invalid rank case: Verify that creating a card with an invalid rank throws a runtime error.

Writing the Deck class

Write the Deck class and store it in a file named **deck.cpp**. You have only one private variable, a vector of Card objects called _deck or deck as you prefer.

- You'll need one constructor, which accepts an optional parameter of type int name cards with a default value of 10. As part of the constructor body, push the number of cards requested onto your private deck vector.

- You'll need an exception named Deck_empty, either inside or outside of class Deck. Remember the "isa" notation we use for exceptions, e.g., **class my_exception : public exception {**...

  You're allowed to declare a class in the public area of Deck – this is called a "nested class". So if you nest Deck_empty inside of Deck, which is a Good Idea, it can be accessed from elsewhere using the membership operator as Deck::Deck_empty.
- You'll need a deal() method that returns the card from the deck vector with the highest index, and deletes it from the vector. But if the vector is empty, throw a Deck_empty exception instead. Again, print nothing!

Writing the test_deck regression test

You'll write a main() function in a file named test_deck.cpp that tests 3 cases:

1. Normative case: Create a deck with the default number of cards, and deal each one into a variable of type Card. If it works, print nothing. If it fails, an exception will be thrown, and you may either catch it and print an error to cerr or allow the test to abort as you prefer.
2. Parameterized Constructor case: Create another deck with MORE than the default number of cards. e.g., 20, and extract them as before. This verifies that the constructor parameter is correct.
3. Failure case: Create a default deck, and deal one more card than the default number of cards that you actually added to the vector. Verify that a Deck_empty exception is thrown; if not, stream an error message to cerr.

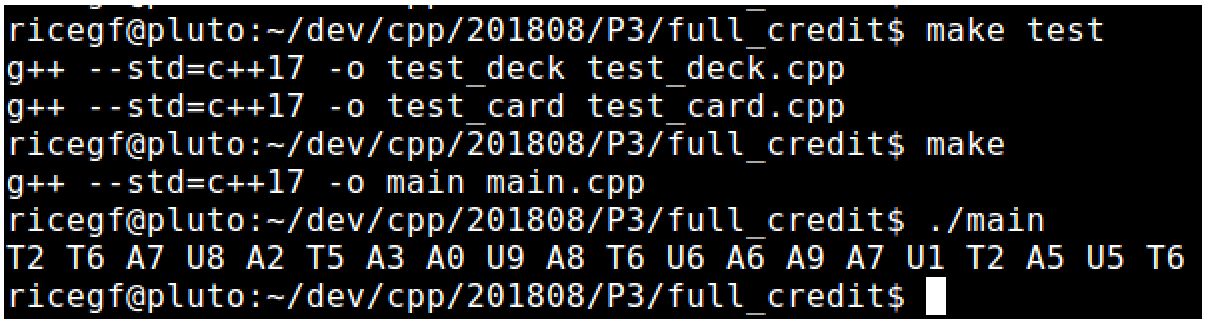
Writing the main application

The main application just creates 20 random cards and prints them to the screen. This is exactly the same as our test_card.cpp first case, so just *move* the code from there. Thus, you'll be back to having no output at all if your regression tests pass!

Deliverables

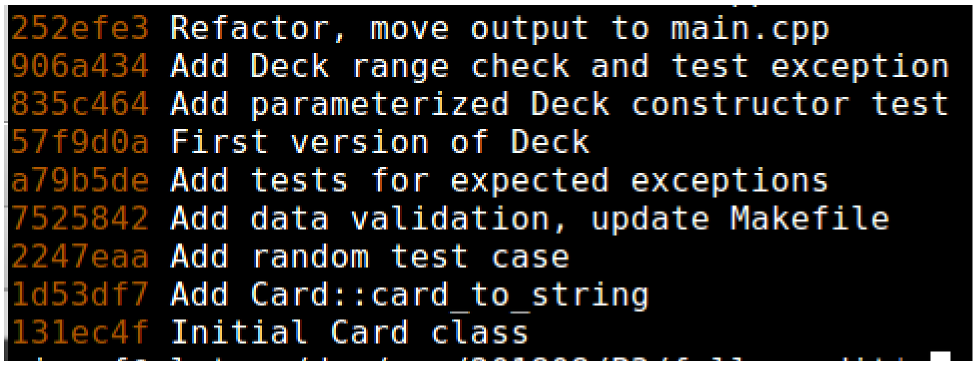
In the full_credit directory, you will provide:

- 5 C++ source files named **card.cpp, deck.cpp, main.cpp, test_card.cpp, and test_deck.cpp**.
- A *required* make file named **Makefile –** I gave you one that should work earlier!
- A screenshot named **main.png** of your interactive testing session similar to mine below.



```
ricegf@pluto:~/dev/cpp/201808/P3/full_credit$ make test
g++ --std=c++17 -o test_deck test_deck.cpp
g++ --std=c++17 -o test_card test_card.cpp
ricegf@pluto:~/dev/cpp/201808/P3/full_credit$ make
g++ --std=c++17 -o main main.cpp
ricegf@pluto:~/dev/cpp/201808/P3/full_credit$ ./main
T2 T6 A7 U8 A2 T5 A3 A0 U9 A8 T6 U6 A6 A9 A7 U1 T2 A5 U5 T6
ricegf@pluto:~/dev/cpp/201808/P3/full_credit$ 
```

- A screenshot named **git.png** of your git log similar to mine below.



```
252efe3 Refactor, move output to main.cpp
906a434 Add Deck range check and test exception
835c464 Add parameterized Deck constructor test
57f9d0a First version of Deck
a79b5de Add tests for expected exceptions
7525842 Add data validation, update Makefile
2247eaa Add random test case
1d53df7 Add Card::card_to_string
131ec4f Initial Card class
```

Bonus (may use some material from Lecture 05)

Continue to use git and a copy of the above code, **refactor your classes to use separate compilation** by splitting the code into a .h (interface) file and a .cpp (implementation) file. Update your Makefile, regression tests, and application to support separate compilation.

As a reminder, if you have a simple class Foo like this:

```
class Foo {
  int _x;
 public:
  Foo(int x = 42) : _x{x} { }
  int getx() {return x;)
};
```

then you can divide it into an interface in a header file foo.h that looks like this:

```
#ifndef __FOO_H
#define __FOO_H
class Foo {
  int _x;
 public:
  Foo(int x = 42);
  int getx();
};
#endif
```

and an implementation in a body file foo.cpp that looks like this:

```
#include "foo.h"
Foo::Foo(int x) : _x{x} { }
int Foo::getx() {return _x;)
```

All of the files that include your Foo class will include the header, not the body. Thus, bar.cpp (which instances a Foo class) would start something like this:

#include "foo.h"
class Bar { ...

You must also change your Makefile to support separate compilation. This one should be close to what you need. Note that to make each .o file, we only need to compile the associated .cpp file – all other classes are included as header (.h) files. Also notice that when we make main, we depend on all of the .o files (so they will only be compiled if their .cpp file has changed), and we only link (only .o files are on the g++ command line).

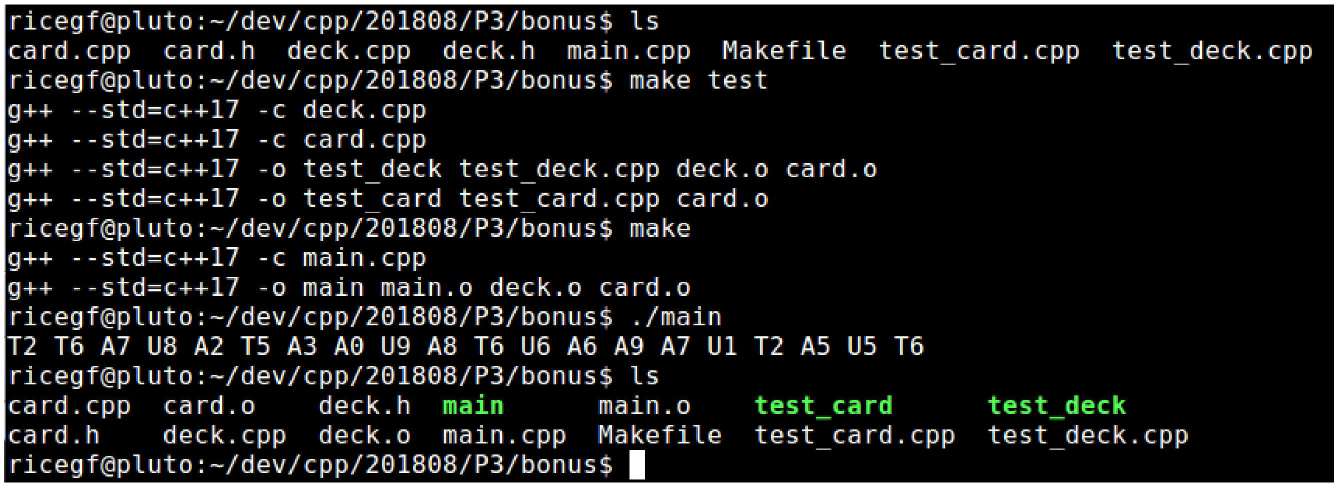
```
# Suggested Makefile for CSE_1325 Homework #3 bonus
CXXFLAGS += --std=c++17

main: main.o card.o deck.o
	${CXX} ${CXXFLAGS} -o main main.o deck.o card.o
test: test_deck test_card
	@./test_card
	@./test_deck
test_deck: test_deck.cpp deck.o card.o *.h
	${CXX} ${CXXFLAGS} -o test_deck test_deck.cpp deck.o card.o
test_card: test_card.cpp card.o *.h
	${CXX} ${CXXFLAGS} -o test_card test_card.cpp card.o
main.o: main.cpp *.h
	${CXX} ${CXXFLAGS} -c main.cpp
deck.o: deck.cpp *.h
	${CXX} ${CXXFLAGS} -c deck.cpp
card.o: card.cpp *.h
	${CXX} ${CXXFLAGS} -c card.cpp
clean:
	-rm -f *.gch *.o a.out test_card test_deck main
```

Deliverables

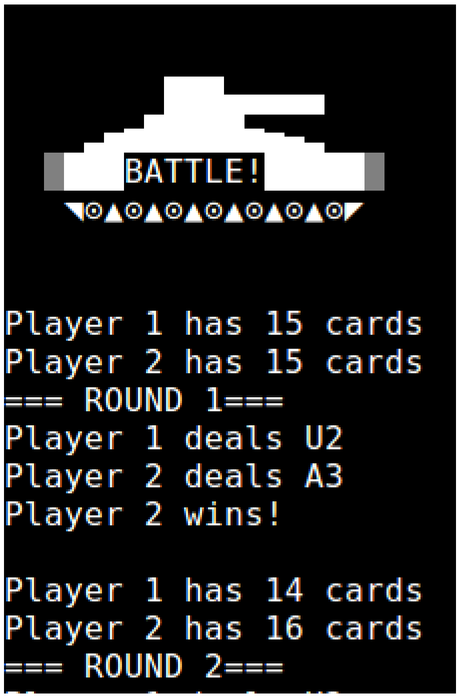
In the bonus directory, you will provide:

- **At least 2 C++ header files** named **card.h** and **deck.h**, respectively.
- At least 5 C++ implementaion files named **card.cpp, deck.cpp, main.cpp, test_card.cpp,** and **test_deck.cpp.**
- A *required* make file named **Makefile**.
- A screenshot named **main.png** of your interactive testing session, as before.
- A screenshot named **git.png** of your git log, as before.



```
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ ls
card.cpp  card.h  deck.cpp  deck.h  main.cpp  Makefile  test_card.cpp  test_deck.cpp
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ make test
g++ --std=c++17 -c deck.cpp
g++ --std=c++17 -c card.cpp
g++ --std=c++17 -o test_deck test_deck.cpp deck.o card.o
g++ --std=c++17 -o test_card test_card.cpp card.o
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ make
g++ --std=c++17 -c main.cpp
g++ --std=c++17 -o main main.o deck.o card.o
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ ./main
T2 T6 A7 U8 A2 T5 A3 A0 U9 A8 T6 U6 A6 A9 A7 U1 T2 A5 U5 T6
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ ls
card.cpp  card.o    deck.h  main      main.o    test_card      test_deck
card.h    deck.cpp  deck.o  main.cpp  Makefile  test_card.cpp  test_deck.cpp
ricegf@pluto:~/dev/cpp/201808/P3/bonus$ 
```

Extreme Bonus



```
Player 1 has 15 cards
Player 2 has 15 cards
=== ROUND 1===
Player 1 deals U2
Player 2 deals A3
Player 2 wins!

Player 1 has 14 cards
Player 2 has 16 cards
=== ROUND 2===
```

Using the Bonus code with our usual array of tools, extend Deck to act like a "real" deck of cards.

For example, when you instance a new Deck without a parameter (the "default constructor", right?), populate it with 30 unique "shuffled" cards by default. Rely on a new public "shuffle" method that you add to Deck, where other programs can access it as well.

Now, write a simple card game (your choice) using the enhanced Deck of 3 suits and 10 ranks. The suggested solution will offer the elementary school favorite of Battle, where 2 players start with half the deck each and flip a card, and the player with the highest rank card returns both cards to the bottom of their deck. Google it if interested, but **you may create ANY game you like.**

You may need other enhancements to Deck or Card to build your game, and anything you like is perfectly acceptable. For example, you may need a way to check if a Deck instance is out of cards. Or perhaps you'll need to return some cards to the top or bottom, or perhaps deal from the bottom (if you're cheatin', as we say in Texas). Add whatever you need.

Update the UML class diagram and your tests as needed to provide a complete implementation.

Deliverables

In the extreme_bonus directory, you will provide:

- **All of your C++ source files**, however many that is.
- A *required* make file named **Makefile**.
- Zero or more screenshots of your interactive game, using any names you like.