Assignment 12: User Defined Data Types: Structures and Enumerated Types

COSC 1437: Programming Fundamentals II

Objectives

- Practice defining and adding our own user defined types to C/C++
- Learn how to use the `enum` enumerated types
- Practice declaring and using `struct` user defined types
- More practice with functions and arrays.

Description

In this assignment we will look at adding our own data types to the C/C++ language by declaring `struct` and `enum` data types. There are basic data types that are defined for you and part of the base C/C++ language. For example, if we need to declare and use a simple integer number, we can create a variable of type `int` like:

```
int myvalue = 42;
```

A declaration like this using a predefined `int` data type causes memory to be allocated somewhere that is big enough to hold and integer value, and in the example shown that memory will be initialized with a value of `42`.

One basic way we can create and add a new data type to the C++ language is to define an enumerated type. We often need to define a set of some small number of options to be used in our programs. Before enumerated types were added to the language, we might have needed to use the C macro preprocessor to create symbolic names, for example:

```
#define GRADE_A 1
#define GRADE_B 2
#define GRADE_C 3
#define GRADE_D 4
#define GRADE_F 5
```

Here we are really assigning a symbolic name and associating it with an integer value. To use defines like this in our program, we would have to use an actual `int` data type, for example:

```
int studentGrade = GRADE_A;
```

This is a bit of an improvement in code readability. Instead of a mysterious 1 we see that the code is assigning a letter grade of A to the student grade variable.

Enumerated types are a much better way of defining a set of related values like this for use in a C++ program. For example, we might create a enumerated type for `Grade` like this:

```
enum Grade {A, B, C, D, F};

Grade studentGrade = A;
```

The `enum` keyword effectively defines a new data type that is added to the C++ language here. So we can now create actual variables of type `Grade` in our programs, and use the more readable names like `A` when referring to grades. It is also now illegal and much harder to assign a nonsensical letter grade to the `studentGrade` variable. Using `#define` we can easily assign some integer, like 0, to a variable meant to hold a letter grade, because the mapping between the integer value and the name is arbitrary. But in an enumerated type the compiler will throw an error if we try and assign anything except the enumerated `Grade` symbols to a `Grade` variable.

We will also be using structures and the `struct` keyword in this assignment to define another type of user defined data type to be added to the C/C++ language. A struct is like a record or a row for a database, it is a collection of data that is related in some way. For example, we might want to have a `Student` data type for an application, defined something like:

```
// add a Student data type to the C/C++ language
struct Student
{
   string firstName;
   string lastName;
   Grade courseGrade;
   double gpa;
};

// allocate memory to hold information about some student
Student aStudent;

// use member access operator to initialize the student record
aStudent.firstName = "Derek";
aStudent.lastName = "Harter";
aStudent.courseGrade = A;
aStudent.gpa = 4.0;
```

We illustrate several things in this code snippet of a `struct` user defined data type. Notice that once declared, we can create variables of type `Student` like we did for the enumerated type before, or like we would for any other built in data type of the language. Notice that we reused the `Grade` enumerated type in our `Student` structure here (you will be doing this in this assignment). Any valid data type can be a member of a structure, including both basic data types of the language, but also other user defined data types that have already been declared.

This example shows how we can use the structure that we declared. We create a variable of type `Student` and we initialize all of its member fields using the member access operator `.`. The type of each member fields is determined by the structure declaration, so `string` members are initialized with strings, the `gpa` member variable is initialized with a `double` value, and the `courseGrade` member variable, which is a `Grade` enumerated data type is initialized with one of the defined enumerated symbols for `Grade`.

In this assignment you will be creating a `Card` structure and use some enumerated types to define the valid face card values (e.g. ACE, TWO, JACK) and suit values (e.g. CLUBS, SPADES). You will write several functions that take user defined data types as inputs, and do things like create arrays of user defined types and operate on them.

Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

File Name	Description
`src/assg12-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`src/assg12-library.cpp`	File that contains the code implementations, all your work will be done here
`include/assg12-library.hpp`	Header include file of function prototypes

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for 'Assignment 12' for our current class semester and section.

2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Lab Task 1 / In-Lab Work : Add a `Suit` Data Type and Implement `suitToString()`

As mentioned in the assignment description, we are going to be building some user defined types and function in order to represent a standard deck of playing cards. We might use the code in this assignment ultimately to build an app to play some kind of card game, like poker or bridge.

To begin with for our task in-lab we will start by defining the enumerated type to represent the `Suit` of the playing cards in a standard deck of cards. The are 4 suits in a standard deck: `DIAMONDS`, `CLUBS`, `HEARTS`, `SPADES`.

NOTE: Our class style requires all user defined data types to start with an initial capital letter, thus your enumerated type should be called `Suit` for this task 1 with an initial upper case `S`. Also class style requires that all enumerated symbols be declared in `ALL_UPPERCASE_NAMES` using under score snake naming convention to separate words in the enumerated symbol.

Declarations of user defined data types should always be placed in a header file. Declare an enumerated type named `Suit` in the library header file for this assignment with the indicated suit types. It is important that the suit types be enumerated in the exact same order as shown above, and have exactly the names shown (in all caps).

As usual you should begin the task by defining the `task1` tests. To test your `Suit` enumerated type, you also need to define and implement a function named `suitToString()`. These functions are helper functions, and we will also use them for testing. In this case, the `suitToString()` function should take a `Suit` as its input parameter. The function returns a `string` from the function. You may find it a bit strange having a function to convert from the `Suit` to a `string`, but it will be useful and make sense as you work on the assignment.

It is suggested that you use a `switch` statement to implement this function. Also note that in the tests we check for an invalid `Suit` being created, so you will need a `default` case in your switch statement that returns the string "SUITUNKNOWN" if somehow an invalid `Suit` is created and passed to this function.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 2: Add a `Face` Data Type and Implement `faceToString()`

Task 2 is basically the same as before, but now you need to declare a `Face` enumerated type. The valid face values for a standard deck of cards are: `ACE`, `DEUCE`, `THREE`, `FOUR`, `FIVE`, `SIX`, `SEVEN`, `EIGHT`, `NINE`, `TEN`, `JACK`, `QUEEN`, `KING`. As before you should declare the enumeration in exactly this order for the `Face` enumerated type, starting with `ACE` and ending with `KING`.

Once you have declared your `Face` enumerated type, define the `task2` tests and implement a `faceToString()` function. This takes a `Face` data type as the input parameter and again returns a `string` result. As with the previous task, you should use a `switch` statement to implement this function, and you should have a `default` case that returns a "FACEUNKNOWN" `string` if an invalid `Face` is passed to your function.

When you are satisfied your function works (make sure your project compiles and runs still), you should commit your changes and push them to the `Feedback` pull request in your classroom repository.

Task 3: Add a `Card` Structure and Implement `cardToString()`

In the third task you will reuse the previous two enumerated types and function to implement a `Card` data type that can be used to represent a single card in a standard deck of cards.

In this task we want to use a `struct` to create a new user defined data type called `Card`. This structure should have 2 members, called `face` and `suit` of type `Face` and `Suit` respectively. Again the declaration of this struct should be put into the `assg12-library.hpp` header file, along with and below the previous two enums that you declared for the `Face` and `Suit`.

Once you have declared your `Card` structure, you should also define the task 3 tests and implement a function named `cardToString()`. This function will take a `Card` data type as input and will return a `string` result. However you need to reuse the previous `suitToString()` and `faceToString()` function in this function. For example, lets say we create a `Card` like this:

```
Card c;
c.face = ACE;
c.suit = SPADES;
```

If we call your function to convert this card to a string, we should get:

```
cout << "The card is: " << cardToString(c) << endl;
```

```
The card is ACE of SPADES
```

You should reuse the `faceToString()` function to get the face `string` and the `suitToString()` function to get the string representation of the suit. Recall that the `string` data type allows you to concatenate strings in C++ using the `+` operator, so for example you can do something like:

```
string faceString = "ACE";
string suitString = "SPADES";
string cardString = faceString + " of " + suitString;
```

to create a new string using the strings returned from the other two function. You will need to access the `face` and `suit` member fields to call these function, and then concatenate the returned strings into a resulting string and return it.

So as a reminder you are required to do the following for this function:

- You are required to pass in a `Card` as input to the function and return a `string` result.
- You are required to reuse the `faceToString()` and `cardToString()` functions when implementing this function.
- You are required to reuse the `Suit` and `Face` enumerated types in the `Card` structure.

- You are required to name the member fields `suit` and `face` (lower case initial letter) respectively.
- You will need to access the member fields `suit` and `face` from the passed in card.
- You will need to concatenate the strings together as shown, putting and " of " string in between the face and suit.

When you are satisfied your function is working, the project still compiles, and you can run and pass the tests, perform the usual to create and push a commit to the `Feedback` pull request.

Task 4: Implement `createDeckOfCards()` function

In this task you will get some more practice performing dynamic allocation of arrays, and you will need to understand how to create and use an array of a user defined data type.

For task 4, we want to create a function that will create an array of `Card` data types and initialize them to represent a standard deck of 52 cards (each of the 13 face values for each of the 4 suits in a deck comes out to 52 cards in a standard deck).

As you should have seen in your materials from this week, we can create an array of a user defined data type, just like we can create an array of basic types. For example, if we want to create an array of 52 cards, to represent a deck of cards, we could do the following:

```
const int STANDARD_DECK_SIZE = 52;

// create an array of 52 cards to represent a deck
Card deck[STANDARD_DECK_SIZE];
```

```
// initialize the first card at index 0 to be
// the "ACE of DIAMONDS"
deck[0].face = ACE;
deck[0].suit = DIAMONDS;
```

A constant named `STANDARD_DECK_SIZE` like the one shown here has been declared for you in the `assg12-library.hpp` header file, you should use it in your code. Here we declare an array of the standard deck size (52) cards, called deck. The array consists of 52 `Card` items indexed from 0 to 51. So as usual we can access any one of these `Card` data types by index. We show accessing the first card at index 0 here in order to initialize its face and suit.

So for task 4, you need to do the following. We need a function named `createDeckOfCards()`. This function does not take any input parameters. We want the deck of 52 cards to be created dynamically. So inside of the function you need to use the `new` keyword to create an array of `STANDARD_DECK_SIZE` number of `Card` structures. This function should return a `Card*` which will be the pointer to the base of the array of cards you dynamically allocate here.

But also we need you to initialize the array of `Card` structures so that all of the 52 different standard cards in a deck are in the array. This means you need to create some loops over the `Suit` and `Face` enumerated types to correctly initialize the deck.

As an example, lets go back to the `Grade` enumerated type we had given previously. If you have done your readings about the enumerated type, you may have run across the following way to iterate over all of the values in an `enum`. For example consider the following code:

```
for (Grade g = A; g <= F; g = static_cast<Grade>(g + 1))
{
    cout << gradeToString(g) << endl;
}
```

Assuming we have an implementation of a `gradeToString()` function that takes a `Grade` as input and returns a string, this example would iterate over the `Grade` enumerated values and print all of them out from "A" to "F". This example is a bit kludgy, we add 1 to the value of g and cast it back to a `Grade` item in order to increment and iterate over all of the values.

You can use the same idea to iterate over the `Suit` enum type and the `Face` enum type. The tests for this function expect that all of the cards for the `DIAMONDS` suit are first in the returned deck of cards, followed by the 13 `CLUBS` then `HEARTS` then `SPADES`. So you need to use a nested loop here. The outer loop should iterate over the suits from `DIAMONDS` to `SPADES`. The inner loop should then iterate over the face card values from `ACE` to `KING`. Then in the inner loop you will cover all 52 combinations of the 4 suits and the 13 faces. You should initialize all of the cards in the array that you dynamically allocated so that you have a standard deck of 52 cards.

Once you have initialized all of the cards in the array representing a deck, you should return the `Card*` that points to the base of the allocated array from this function.

So the suggested steps to implement the `createDeckOfCards()` function are as follows:

1. Dynamically allocate an array of `Card` structures. You should call this array `newDeck`. Use the defined constant `STANDARD_DECK_SIZE` to allocate an array that will contain 52 cards.
2. Create an integer variable called `index` initialized to 0 before your loops.
3. Create nested for loops.
    - The outer for loop should iterate over the suits from DIAMONDS to SPADES.
        - The inner for loop should iterate over the faces from ACE to KING
            * Inside of the two loops you need to initialize each `Card` structure in the dynamically allocated array.
            * The `index` tells you which card in your `newDeck` array you need to initialize. The loop variables will tell you the value of the suit and face to initialize this card to.
            * Increment the `index` by 1 after you initialize the indicated card.
4. After initializing the cards, return the pointer to the dynamically allocated base address of the array of `Card` you created.

You will see that the tests of task 4 expect the cards to be initialized exactly in the order specified. So for example the card at index 0 needs to be the "ACE of DIAMONDS", and the card at the last index 51 will be the "KING of

SPADES".

When you are satisfied your function is working, the project still compiles, and you can run and pass the tests for this task, perform the usual to create and push a commit to the `Feedback` pull request.

Task 5: Implement `shuffleDeckOfCards()` Function

For our final task, lets write another function that will randomly shuffle our deck of cards. This would be a necessary prerequisite for a card game before beginning to play.

Create a function named `shuffleDeckOfCards()`. This function will be a `void` function this time, it does not return an explicit result. You will pass in an array of `Card` items that represents a standard deck of cards. This function can assume that there are 52 cards in the `Card` array, so we will not pass in the size of this array as we have done in previous assignments. Also since, as you should know, arrays are basically passed by reference, when we shuffle the cards in the deck array given to this function, the caller will see a shuffled deck of cards after it returns.

We will use the `rand()` function from the C standard library to perform some random shuffling in this function. The `<cstdlib>` is already included for you in the `assg12-library.cpp` file so that you can call this function. We are going to shuffle the deck of cards by doing the following:

1. Iterate over all of the cards in the array from index 0 to index `STANDARD_DECK_SIZE - 1`
    - Randomly generate a number in the range from 0 to 51 to serve as the index of another card in the deck.
    - Swap the card at the index we are iterating with the card at the index we randomly generated.

To help you along, we will give a few examples of how to implement some of the things you need for this function. To generate a random card index, do something like the following:

```
int randomIdx = rand() % STANDARD_DECK_SIZE;
```

This will generate a random number in the range from 0 to `STANDARD_DECK_SIZE - 1`, which is what you need. By performing the modulus we get the remainder from dividing by 52 (the standard deck size), the remainder from an integer division gives a number in the range from 0 to 51 here. The `rand()` function returns a random positive integer, so the modulus changes the random number into the range we need for this task.

Likewise you can swap two cards in the array in the same way you have done before. For example, if I have and `idx` and a `randomIdx` I want to swap, I can perform the following:

```
Card tmpcard = deck[idx];
deck[idx] = deck[randomIdx];
deck[randomIdx] = tmpcard;
```

Basically assignment works for user defined structures like it does for regular data types. So we can swap a `Card` in the same way we would swap any other data type.

As a final note for task 5, notice that we do test your function to randomly shuffle a deck of cards. We do this by setting the random seed using the `srand(42)` call. What this does for a random number generator is that after seeding, it will generate the same sequence of random numbers. So if you generate a random index as shown using `rand()` and then swap the cards in the array as shown, you should get the same random shuffle that the test for this task expects.

When you are satisfied your function is working, the project still compiles, and you can run and pass the tests, perform the usual to create and push a commit to the `Feedback` pull request.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 50 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 25 additional points are given by the autograder for completing tasks 2 and 3 successfully.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax