EECE 1080C
Programming for ECE
Assignment 3

Sina Eghbal, Petar Acimovic
(eghbalsa@mail.uc.edu, acimovpr@mail.uc.edu)

March 13, 2023

1 Instructions

- **Dute Date**: Sunday, March 26th 2023 by 11:59 PM.
- **Late Penalty**: 10% penalty for each day, up to 3 days. Hard deadline on Wed, March 29th at 11:59 PM.
- Submit your solution to each question in a separate file. Make sure you put your name as a comment in every cpp file, compress all the files into an archive file, and upload it on Canvas.
- If you have any questions, please post it on Canvas.
- This is an individual assignment. You can discuss your approach on how to tackle the problems, but you should **not** share your code with anyone.
- You may be asked to explain your work, and your grade may change based on your explanation.
- If you are found to have cheated, academic misconduct procedure will be initiated.
- Make sure your code is well-indented, your variables have meaningful names, and you leave enough comments in your code so that there is no confusion for those who read your code as to what it does.
- There might be some test cases in each question. But that does not mean those are the only tests that your program should pass. Your code will be looked at thoroughly, and will be tested against certain edge cases.
- You **should not** use any tools that have not yet been covered in this course yet, in particular **pointers**.

2 Questions

Q0 Create a class *Point* with two private member variables, x and y of type *double*. Create two getter functions, and two setter functions called *get_x()*, *set_x(double x)*, *get_y()*, *set_y(double y)* to set and get the values of x and y.
Create an **abstract class** called Shape with seven constant **pure virtual** functions *area()*, *perimeter()*, *get_min_y()*, *get_max_y()*, *get_min_x()*, *get_max_x()*, and *get_name()*. Create three constant member function called *print_extremums()*, *print_area()*, and *print_perimeter()* returning nothing but printing the values of extremum points, the area, and the perimeter in the following format:

```
print_area();
// Area: {area}
print_perimeter();
// Perimeter: {perimeter}
print_extremums();
// Max x: {max_x}
// Min x: {min_x}
// Max y: {max_y}
// Min y: {min_y}
```

Let *Circle* and *Rectangle* be two classes deriving from *Shape*. *Circle* should have two member variables; one of type *Point* representing its centroid, and the other of type *double* representing its radius. It should also have two constructors; one taking 3 *double*s, and the other taking a *Point* and a *double*. If the value passed as the radius is negative, negate it before assigning it to its respective member variable.
Rectangle should have two member variables of type *Point* representing its top-left and bottom-right corners. A *Rectangle* should have two constructors. One with 2 *Point*s and the other with 4 *double*s representing x_1, x_2, y_1, and y_2. Keep in mind that regardless of what x has been associated with what y in the two *Point*s passed to the constructor, the two *Point*s in *Rectangle* should store its top-left and the bottom-right coordinates.
Implement all the pure virtual functions for the subclasses. Note that *get_name()* should return "Rectangle" for instances of *Rectangle*, and "Circle" for objects of type *Circle*.
Create a stand-alone function called *print_shape(...)* that takes a *Shape* and prints its name, its area, its perimeter, and its extremum points.
Test what you have written with the following code, and verify the output of your program manually.

```
Point centroid(-13.0, 17.0);
Circle circle1(centroid, 19.0);
Circle circle2(18.0, -19.0, -17.0);
Point top_right(10, 0); Point bottom_left(5, -12);
Rectangle rect1(top_right, bottom_left);
Rectangle rect2(5, 10, 0, -12);
print_shape(circle1);
print_shape(circle2);
print_shape(rect1);
print_shape(rect2);
```

[1 point]

Q1 Create a class called *Time* with the following 3 member variables:

- second : unsigned char
- minute : unsigned char
- hour : unsigned char

Let your class have 3 different constructors. A default constructor that initialises all member variables to 0, a constructor taking 3 *unsigned char*s and assigning them to their corresponding member variables, and the third one taking a string of the format "HH:MM:SS" (may be appended by AM/PM - where PM is to be ignored if HH > 12, and AM is to be ignored if HH < 12), parsing it and assigning its values to its corresponding member variables. Note that for the third constructor you might have to use *std::string::find(...)*[1] and *std::to_string(...)*[2].
Your class should not store invalid times (eg. 00:60:00, 24:00:00, or 00:00:60) as they are passed to its constructors. Instead, it should convert them into valid time values (eg. 00:60:12 should be stored as 01:00:12 , 24:00:00 as 00:00:00, and 00:60:63 as 01:01:03).
Overload the operators $+$, $-$, $>$, $<$, and $==$ for your class. It is clear that $01 : 01 : 03 + 02 : 05 : 04$ should return $03 : 06 : 07$. But for a case like $02 : 07 : 03 + 23 : 54 : 59$, your program should return $02 : 02 : 02$.
Create a *to_string()* function that takes a boolean (give your boolean a default value of your choice). Your function should convert the current values to a string of the format $HH : MM : SS$. If the argument is *true*, the function must return a string that stores the time in the 12-hour format (AM or PM should be displayed after SS). Otherwise, the function must return the time in the 24-hour format.
Test your class by creating a few instances, with legal, and illegal values. Make sure you test your code with a few cases which involve value overflows (eg. minute ≥ 60), and underflows (eg. *t1 - t2* where *t1.second* < *t2.second*) at construction, when adding two *Time*s together or subtracting them from each other. Verify that your comparison operators work as expected. Test your *to_string(...)* function with the flag set to *true* and *false*.

```
Time t1;
Time t2(11, 06, 61);   // What should happen to 61?
Time t3("13:67:73");   // Input of type string,
// again to be stored differently from what it looks like
t2.to_string(true);    // 11:07:01AM
```

[1]https://en.cppreference.com/w/cpp/string/basic_string/find
[2]https://en.cppreference.com/w/cpp/string/basic_string/to_string

```
t3.to_string(false);   // 14:08:13
t1 = t2 - t3;          // 20:58:48
Time t3 = t2 + t3;     // 01:15:14
t2 < t21               // false
```

[2.5 points]

Q2 We have mentioned that a string is an array of characters. *string* is not a native data type in C++ (and that is why it is part of the *std* namespace). In this question we are going to be creating our own class to represent strings from scratch (without using any of the *std::string* functions or even including it).

Create a class template called *StringT* that contains an array of characters of length n where n is a template parameter. Regardless of the length of the array, a string will be ended (and nothing will be displayed or read) after it reaches a '\0'[3]. Let your class have a member function called *size()* that returns the length of the string (The minimum of the length of its underlying array and the number of characters before '\0'.).

Overload the =, ==, +, + =, >, and < operators for your string class.

Create a member function called *substring(unsigned int start_idx, unsigned int end_idx)* that returns the substring beginning at *start_idx* and ending right before *end_idx*. Your function should be able to handle unexpected inputs (eg. *end_idx* < *start_idx*, or *end_idx* >= *size*).

Create one more member function called *reverse()* that returns the reverse of the current string, and two other member functions called *find(...)*, one taking a *char* and the other taking a *StringT*. Both functions should return the index of the first occurence of their argument in the original string (the starting index if the argument is of type *StringT*) if it exists, and -1 otherwise.

Create two other member function called *is_empty()*, *clear()*. Overload the +, + =, and [n] operators to concatenate and return, concatenate the RHS string to the one on the LHS, and get the $n + 1^{th}$ character of the string.

```
String my_string<1000> = "this is my string";
cout << my_string.arr << endl; // "this is my string"
cout << my_string.substring(1, 4).arr << endl; // "his"
cout << my_string.reverse().arr << endl; // "gnirts ym si siht"
cout << my_string[3] << endl; // s
cout << my_string.find('i') << endl; // 2
String<1000> is_string = "is";
cout << my_string.find(is_string) << endl; // 2
String<1000> ing_string = "ing";
cout << my_string.find(ing_string) << endl; // 14
```

[3 point]

Q3 Write a class to represent arrays of type T. Arrays represented in your program may have one, two, three, or four dimensions. Dimensions are

[3] '\0' is referred to as the end-of-string character.

initially specified using template arguments (and default values may be used for d-2, d-3, and d-4). Regardless of the dimension of your array, its underlying data structure must be a 1-d array. Create a constructor that sets all elements of your array to a default value, and another one that takes a C++ array of the same size as your array. Create a function called *reshape(..)* that changes the number of dimensions, and the number of elements in each dimension of the array. For instance, a 1-d array of length 12, can be reshaped to have two dimensions of lengths (3, 4) or (2, 6), or 3 dimensions of lengths (2, 2, 3). Create two *get(...)* functions, one taking a reference to an element of type T, and the other taking a reference to a sub-Array. Let both of your functions return booleans indicating the success or failure of the *get(...)* functions. Let your class have two *set(...)* functions. one for assigning to a dimension, and the other for assigning to a particular element. Overload the == operator that checks for matching dimensions and matching elements in the underlying 1-d array.

```
bool b;
Array<int, 6> int_arr({2, 3, 5, 7, 11, 13}); // A single d array of length 6
b = int_arr.set(3, 17); // {2, 3, 5, 17, 11, 13}, b = true
b = int_arr.reshape(4, 2); // b = false, no change
// since an array of 6 cannot be 4 * 2.
// The underlying array doesn't change but
int_arr.reshape(2, 3); // Accessing the array will
// {{2, 3, 5},
// {17, 11, 13}}
int_arr == Array<int, 2, 3> ({2, 3, 5, 17, 11, 13}); // true
b = int_arr.set(3, 19); // b = false since the first dimension
// is itself an array (and there are two such sub-arrays)
b = int_arr.set(1, 1, 23); // b = true
// {{2, 3, 5},
// {17, 23, 13}}
Array<int, 3> sub_arr({-1, -1, -2});
int_arr.set(1, sub_arr);
// {{2, 3, 5},
// {-1, -1, -2}}
int_arr.reshape(3, 2);
// {{2, 3},
// {5, -1},
// {-1, -2}}
b = int_arr.get(0, sub_arr) // b = true
// sub_arr = {2, 3, -2} // The last element didn't change
```

[**2.5 points**]

Q4 Use the following interface to create a class that implements a simple game where a player on an 8×8 grid starts from the left bottom corner of the grid. When the run function is called, the game should wait for the user input. The user should be able to use the arrow keys (or W, A, S, and D as alternative keys) to move the player on the board. You should implement your game so that when the player tries to move beyond the grid, the move is ignored. Once the user makes a move, the program should reprint the board so that it reflects the new location of the player. The program should terminate if the user takes the player to the top right cell in the grid or if the user presses any keys other than those 8 control keys. Use the following interface to implement the game.

```cpp
enum class dir {up, down, right, left};
class GridWorld {
protected:
        unsigned char player_x;
        unsigned char player_y;

        // Invalidate the previous terminal by clearing
        // it and reprinting the new state of the board.
        virtual void invalidate();
        // If the move is legal move the player.
        virtual bool move(dir d);

public:
        GridWorld() : player_x(0), player_y(0) {}
        virtual void run();
};
int main() {
        GridWorld gw;
        gw.run();
        return 0;
}
```

To capture a single user input from the user without it appearing on the terminal use the *getch()* function (part of conio.h on Windows, and contact me if you need help on Mac or Linux). You can use *gameoflife.cpp* as a starting point. [**2 points**]

Q5 Suppose we are using a 2-d array to represent a grid. Each cell in the grid may be blocked (marked with X), or not-blocked. Represent the following grid using a 2-d array. There is a single player placed at point S on the grid. The player will be able to move through the cells that are not blocked horizontally or vertically. Derive from *GridWorld*, and implement the game so that it can navigate in those cells that are not blocked.

			X				
X		X	X		X	X	
			X			X	X
	X			X			
		X		X	X	X	
	X			X			
	X		X	X		X	
S	X					X	

```cpp
class GridWorldBlocks : public GridWorld {
        bool board[w][h];
protected:
        virtual bool move(dir d);
        virtual void invalidate();

public:
        GridWorldBlocks(bool _board[w][h]) {}
};
```

You should be able to implement the modified game using the above interface. Remember that you can reuse the code written in *GridWorld::move(...)* by calling it from within *GridWorldBlocks::move(...)*. [**1.5 points**]