Name: ____ ID: ____________________

Object-Oriented Programming Lab #2 **Jan 27th, 2023**

Working with C++ Functions

1. Write functions to build pattern strings with varying sizes. All functions **must take** the pattern size as an input and **must not** contain trailing spaces before the end of each line.

1.1) Write a function **`tri_pattern`** to build a pattern string like shown below:

`auto s = tri_pattern(0);`
`std::cout << s;`
	`auto s = tri_pattern(3);`
	`std::cout << s;`
		`auto s = tri_pattern(5);`
		`std::cout << s;`

	`*`
	`**`
	`***`
		`*`
		`**`
		`***`
		`****`
		`*****`

1.2) Write a function **`arrow_pattern1`** to build a pattern string like shown below:

`auto s = arrow_pattern1(0);`
`std::cout << s;`
	`auto s = arrow_pattern1(2);`
	`std::cout << s;`
		`auto s = arrow_pattern1(4);`
		`std::cout << s;`

	`*`
	`**`
	`*`
		`*`
		`**`
		`***`
		`****`
		`***`
		`**`
		`*`

1.3) Write a function **`arrow_pattern2`** to build a pattern string like shown below:

`auto s = arrow_pattern2(2);`
`std::cout << s;`
	`auto s = arrow_pattern2(3);`
	`std::cout << s;`
		`auto s = arrow_pattern2(5);`
		`std::cout << s;`

` *`
`**`
` *`
	`  *`
	` **`
	`***`
	` **`
	`  *`
		`    *`
		`   **`
		`  ***`
		` ****`
		`*****`
		` ****`
		`  ***`
		`   **`
		`    *`

1.1) ☐	**1.2)** ☐	**1.3)** ☐

2. Given the following **`main`** function:

```
int main()
{
    auto words = {"C", "**", "*C++*", "*Java", "*Python*", "Rust*"};
    for (const auto& w: words) {
        std::cout << unstylize(w) << std::endl;
    }
}
```

2.1) Write the function **`unstylize`** which is used by the **`main`** function above to remove an enclosing `"*…*"` pairs from a word and complete the program so that it prints the word "C", "", "C++", "*Java", "Python", and "Rust*" on the screen.

2.2) Write the function **`stylize`** which is used to enclose `"<strong>…</strong>"` around a word and **rewrite** the **`main`** function above to use **`stylize`** instead and complete the program so that it prints the output as shown on the right:

Program Output (for 2.2)

```
C
<strong></strong>
<strong>C++</strong>
*Java
<strong>Python</strong>
Rust*
```

2.3) Write a program that read words from standard input and generate a table in HTML format that present the data as shown on the output table below (adjust the CSS style as needed).

Input

```
C *C++* Rust*
*Python* * *Java
```

Output

```
            unstylized   stylized
C           C            C
*C++*       C++          <strong>C++</strong>
Rust*       Rust*        Rust*
*Python*    Python       <strong>Python</strong>
*           *            *
*Java       *Java        *Java
```

Hint: For string **`s`** (`std::string`), you can use **`s.substr(i, n)`** for getting a substring of **`s`**, **`s.front()`** and **`s.back()`** for getting the first and the last character.

2.4) Partition the program from 2.1) to 2.3) to have **at least** one header file for all utility functions used by the program, one source file for definitions of all utility functions, and one source file for each program. **Create CMakeLists.txt** file for the project, configure and build all programs specified in the project. Finally, test the programs by running all of them.

Advice: Use I/O redirection to avoid typing the same input over multiple runs and save the output to a file for later read.

2.1)	☐	**2.2)**	☐	**2.3)**	☐	**2.4)**	☐

3. Given the following SVG image file as a template:

```
<svg width="500" height="500" xmlns="http://www.w3.org/2000/svg">
  <rect width="100%" height="100%" fill="#EEEEEE" />
  <circle cx="250" cy="250" r="250"
          stroke="black" stroke-width="2" fill="none" />
  <circle cx="250" cy="250" r="10" fill="#00FFFF" />
</svg>
```

3.1) Write a program that:

- Take the number `N` from user input and use it to generate `N` points $p_i = (x_i, y_i)$
  With $-1 \le x_i \le 1$ and $-1 \le y_i \le 1$
- Map the point to draw a circle in the SVG image output (print <circle> element), use different fill colors and circle sizes for the point inside the unit circle and the point outside

3.2) Partition the program from 3.1) to have **at least** one header file for all utility functions used by the program, one source file for definitions of all utility functions, and one source file for the program. **Create CMakeLists.txt** file for the project, configure and build the program. Finally, test the program.

Advice: Use I/O redirection to save the output to a file for viewing from the browser.

3.1)	☐	**3.2)**	☐