Name: _____ ID: _____

Object-Oriented Programming Lab #7 **Mar 3rd, 2023**

Review Exercises

1. Write functions and programs that manipulates various data structures. Verify correctness of all functions with test programs and ensure that all programs run as expected.

1.1) Write a function, `substitute(text, word, rep)`, that creates a new string with replacements of all substring matched by `word` in `text` with `rep`.

Example test cases

```
std::string text1 = "abc python javapythonx";
std::string text2 = "abcx yja helx yz01 23";

substitute(text1, "python", "rust") == "abc rust javarustx"
substitute(text1, "", "rust")       == "abc python javapythonx"

substitute(text2, "python", "rust") == "abcx yja helx yz01 23"
substitute(text2, "x y", "a b")     == "abca bja hela bz01 23"
```

1.2) Modify `substitute` from **1.1)**, by requiring `text`, `word`, and `rep` to be a C-style string and return a new C-style string from the function. **Do not use** any standard library functions. **Do not use** subscript operator in the function body.

1.3) Write a function, `create_pairs(arr)`, that creates pairs of numbers from a number list. Discard the remaining value that cannot form a pair.

Example use case

```
vector<pair<double, double>>
    create_pairs(const vector<double>& arr);

vector<double> data{
    1.2, 2.3, 5, 2, 1.1, 3, 7
};
auto pairs = create_pairs(data);
```

Expected object value

```
pairs: {
    {1.2, 2.3}, {5, 2},
    {1.1, 3}
}
```

1.4) Modify `create_pairs` from **1.3)**, by taking a C-style array as parameters instead. Specify an appropriate return type by defining additional support types as necessary. **Do not use** any standard library functions and classes.

1.1)		1.2)		1.3)		1.4)	

2. Write functions and programs that manipulates ragged arrays. Verify correctness of all functions with test programs and ensure that all programs run as expected.

2.1) Given the following program as a template:

```cpp
#include <vector>

template <class T>
using Vec = std::vector<T>;

template <class C>
Vec<Vec<double>> create_ragged_array(
    const C& c, double val)
{
    Vec<Vec<double>> out;
    for (auto n: c)
        out.emplace_back(n, val);
    return out;
}
```

```cpp
#include <iostream>

int main()
{
    auto dims = {2, 5, 3};
    auto rg_arr
        = create_ragged_array(
              dims, 1.3);
    for (const auto& v: rg_arr) {
        for (auto x: v)
            std::cout << ' ' << x;
        std::cout << '\n';
    }
    std::cout << std::endl;
}
```

- Modify `create_ragged_array` so that it creates an equivalent dynamic ragged array with the same dimensions **without** using any standard library functions and classes
- Modify the main function as needed for printing the data
- Do not worry about memory exhaustion

2.2) Extend the program From **2.1)** by adding the following functions:

- `copy_ragged_array(data)`: creates a ragged array with contents from `data` which is `Vec<Vec<double>>`
- `del_ragged_array(arr)`: free all memory owned by a ragged array structure `arr`

Do not use any standard library functions and classes except for the given input parameters.

*Example use case for **2.2)***

```
int main()
{
    std::array<std::size_t, 3> dims{2, 6, 4};
    Vec<Vec<double>> data{
        {1.2, 2.3},
        {5, 2, 1.1, 3, 7},
        {4.2, 2.3, 3.5}
    };

    auto rg_arr1 = create_ragged_array(dims, 0.5);
    auto rg_arr2 = copy_ragged_array(data);

    // print rg_arr1, rg_arr2

    del_ragged_array(rg_arr2);
    del_ragged_array(rg_arr1);
}
```

Expected object value

```
rg_arr1: {
    {0.5, 0.5},
    {0.5, 0.5, 0.5, 0.5, 0.5,
0.5},
    {0.5, 0.5, 0.5, 0.5}
};

rg_arr2: {
    {1.2, 2.3},
    {5, 2, 1.1, 3, 7},
    {4.2, 2.3, 3.5}
};
```

2.3) Extend the program From **2.2)** by adding the following functions:

- `copy_ragged_array(data)`: creates a ragged array with contents from `data` which is another ragged array created by `create_ragged_array` or `copy_ragged_array` (`Vec<Vec<double>>` version)

2.4) Defining a class `Ragged_array`, that can be used as a substitute for the structure in **2.3)** with appropriate constructors (including default and copy constructors), destructor, assignment operator, and member functions. Rewrite the program from **2.3**) using `Ragged_array`.

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

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

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <polyline fill="none" stroke="black"
    points="10, 10 190, 20 180, 180 80, 110 10, 10"
  />
</svg>
```

3.1) Defining a class `Poly_line`, that stores a list of points (x_i, y_i) (assumed to be integers) forming line segments along with its appropriate operations.
You are required to:

- Provide appropriate **constructors** for class `Poly_line`
- Provide appropriate **member functions** for accessing the points within the `Poly_line`
- Provide a **non-member function**, `print(segs, os)` to print the contents of a `Poly_line` `segs` to the output stream `os`
- Provide a **non-member function**, `gen_svg(segs, os)` to generate an SVG representing the line segments from the contents of `segs` to the output stream `os`
- Write a test program for testing all use cases of a `poly_line` object and its operations including the test for constructing `Poly_line` object, passing `Poly_line` to a function, returning `Poly_line` from a function, constructing a `Poly_line` from another `Poly_line`, and copying a `Poly_line` object.

3.2) Modify `Poly_line` from **3.1)** to store its contents using free store memory **without using the C++ standard library**.

3.1)	**3.2)**