Object–Oriented Programming
Lab 11: On Chapter Templates
ENSIA 2024-2025

Exercise 1

The file « pair.h » contains the definition of a class Pair, which (at the moment) holds a pair of int. The function print_pair() can be used to print the values that the pair contains.

```
class Pair
{
public:
    Pair(const int& t, const int& u): first(t), second(u){}
    int& get_first() { return first; }
    int& get_second() { return second; }
    const int& get_first() const { return first; }
    const int& get_second() const { return second; }
private:
    int first;
    int second;
};

void print_pair(const Pair& p)
{
    std::cout << '(' << p.get_first() << ", " << p.get_second() << ")\n";
}
```

1)

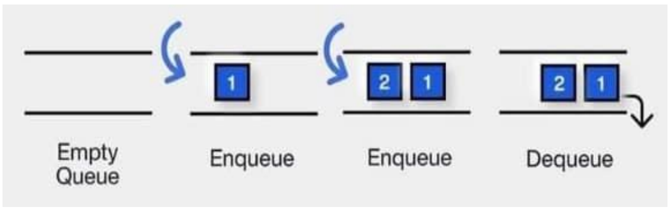
- **a)** Change the definition of Pair so that it becomes a *class template*. Rather than a pair of int, it should be able to hold any two objects of (possibly) different types.
- **b)** After making this change, note that print_pair() generates a compile error. Fix this error by turning print_pair() into a *function template* to print any kind of generic Pair
- **c)** In main.cpp, test your Pair and print_pair() implementations by creating a Pair of int, a Pair of doubles, and a Pair containing a std::string and an int.

2)

- **a)** At the moment, all the member functions of Pair are defined inside the class. Move the definitions of the constructor and get functions so that they are declared inside the class, but defined outside (but still in the header)
- **b)** Ensure that your main function compiles correctly and gives the expected results.

Exercise 2:

A queue is a first-in, first-out structure. An example of a queue is a line of people waiting to be served. We can implement a queue using an array, in which the items are inserted (enqueued) at the end of the array and removed (dequeued) from the front. Create a template class and a queue of any type. Then test the queue class with two specializations, one as a queue of integers, the other as a queue of strings.



```cpp
template <class T>
class Queue
{
private:
    int Qsize;              // maximum number of elements in the queue
    int last;               // location of the last element "back"
    T* queuePtr;

public:
    Queue(int = 10);        // default constructor (queue size 10)
    ~Queue();
    bool push(const T&);
    bool pop(T&);
    bool isFull() const;
    bool isEmpty() const;
    void print() const;
};
```

Exercise 3:

1.

- Create a class **DoubleSubscriptedArray** that has similar features to the class **Array** in Figs. 11.6-11.7. At construction time, the class should be able to create an array of any number of rows and any number of columns.
- The class should supply an **operator()** to perform double-subscripting operations. E.g., in a 3-by-5 **DoubleSubscriptedArray** called **a,** the user could write **a(1, 3)** to access the element at row 1 and column 3. The underlying representation of the double-subscripted array should be a single-subscripted array of integers with **rows * columns** number of elements.

  The function **operator()** should perform the proper pointer arithmetic to access each element of the array. There should be two versions of **operator()** -- one that returns **int &** (so that an element of a **DoubleSubscriptedArray** can be used as an lvalue) and one that returns **const int &** (so that an element of a const **DoubleSubscriptedArray** can be used only as an rvalue).
- The class should also provide the following operators: ==, !=, =, << (for outputting the array in row and column format), and >> (for inputting the entire array contents). By default, the array has 10 * 10 elements.

2.

- Write a driver to test the various member functions; the following is a sample output:
  Uninitialized array “**a**” is:

  ```
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  ```

  Uninitialized array “**b**” is:

  ```
  0 0
  0 0
  0 0
  0 0
  ```

```
0 0
0 0
0 0
0 0

Initialized array “a” is now:
89 13 54 19 27 30 76
96 22 21 7 97 26 43
64 90 4 86 6 66 92
92 86 19 80 76 60 49
84 15 56 5 39 97 53
12 34 49 91 54 98 39

Assigning b = a :
89 13 54 19 27 30 76
96 22 21 7 97 26 43
64 90 4 86 6 66 92
92 86 19 80 76 60 49
84 15 56 5 39 97 53
12 34 49 91 54 98 39

“a” was found to be equal to “b”
The element (2, 1) of array “a” is: 90

Changed element (2, 1) to -1 :
89 13 54 19 27 30 76
96 22 21 7 97 26 43
64 -1 4 86 6 66 92
92 86 19 80 76 60 49
84 15 56 5 39 97 53
12 34 49 91 54 98 39

“a” was found not to be equal to “b”
```

- Create a template of the **DoubleSubscriptedArray** using a type parameter **elementType** when defining this class. Name your template class **Table**. This template enables **Table** objects to be instantiated with a specified element type at compile time. The underlying representation of the double-subscripted array should be a single-subscripted array with rows * columns number of elements.
- Create a driver to test the capabilities of your class. The following is a sample output:

  <u>Sample Output:</u>

```
Uninitialized array “a” is:
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
Uninitialized array “b” is:


Initialized array “a” is now:
89 13 54 19 27 30 76
96 22 21 7 97 26 43
64 90 4 86 6 66 92
92 86 19 80 76 60 49

Initialized array “b” is now:
```

```
g g
g g
g g
g g
g g
```

```
Enter values for b (10 of them):
a b c d e f g h i j
a b
c d
e f
g h
i j
```

```
The element (2, 1) of array "a" is: 90
Changed element (2, 1) to -1 :
89 13 54 19 27 30 76
96 22 21 7 97 26 43
64 -1 4 86 6 66 92
92 86 19 80 76 60 49
```