Exercise 1

Write a `SafeArray` class that use dynamic array to store integers in a fixed-size array. The class should have the following methods:

- `SafeArray(int sz)`: constructor that initializes the array to size `sz` containing all 0 values.
- `SafeArray(const SafeArray& other)`: copy constructor that creates a new `SafeArray` object that is a deep copy of `other`.
- `int& operator[](int i)`: returns the reference to the element at index `i`. It throws an exception of type `std::out_of_range` if the index is out of range.
- `int size()`: returns the size of the array.
- `void print()`: prints the elements of the array.
- `~SafeArray()`: destructor that frees the memory allocated for the array.

Demonstrate the use of the `SafeArray` class in the `main` function.

```
int main() {
    SafeArray a(5);
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    a[3] = 4;
    a[4] = 5;
    a.print();
    cout << "Size: " << a.size() << endl;
    SafeArray b = a;
    b.print();
    cout << "Size: " << b.size() << endl;
    b[0] = 10;
    b.print();
    cout << "Size: " << b.size() << endl;
    try {
        b[5] = 10;
    }
    catch (std::out_of_range& e) {
        cout << "Caught an exception of type: " << e.what() << endl;
    }
}
```

Exercise 2

Derive a class `SafeVector` from the `SafeArray` class that uses resizable array to store integers. Add the following methods to the `SafeVector` class:

- `void push_back(int val)`: adds a new element `val` at the end of the vector.
- `void pop_back()`: removes the last element from the vector. The function should throw an exception of type `std::underflow_error` if the vector is empty.
- `int back()`: returns the last element of the vector.

Which member functions need to be virtual in the `SafeArray` class? Which needs to be overridden in the `SafeVector` class?

Write a `main` function to demonstrate the use of the `SafeVector` class. Include the case when the vector is empty and the `pop_back` function is called.

Exercise 3 .....

In the C++ exception mechanism, control moves from the **`throw`** statement to the first **`catch`** statement that can handle the thrown type. When the **`catch`** statement is reached, all of the automatic variables (i.e., local and argument variables) that are in scope between the **`throw`** and **`catch`** statements are destroyed in a process that is known as *stack unwinding*.

```
class MyException{};

class Dummy {
    public:
    string MyName;
    int level;
    void PrintMsg(string s) {
        cout << s  << MyName <<  endl;
    }
    Dummy(string s) : MyName(s) {
        PrintMsg("Created Dummy:");
    }
    Dummy(const Dummy& other) : MyName(other.MyName) {
        PrintMsg("Copy created Dummy:");
    }
    ~Dummy() {
        PrintMsg("Destroyed Dummy:");
    }
};

void C(Dummy d, int i) {
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i) {
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i) {
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    //  Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
```

```
    //   delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main() {
    cout << "Entering main" << endl;
    try {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e) {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}
```

The output of the program is:

```
Entering main
Created Dummy: M
Copy created Dummy: M
Entering FunctionA
Copy created Dummy: A
Entering FunctionB
Copy created Dummy: B
Entering FunctionC
Destroyed Dummy: C
Destroyed Dummy: B
Destroyed Dummy: A
Destroyed Dummy: M
Caught an exception of type: class MyException
Exiting main.
```

(a) What is the order in which the `Dummy` objects are created and then destroyed as they go out of scope.?

(b) Which functions completed their execution in the above program?

(c) Uncomment the definition of the `Dummy` pointer and the corresponding **`delete`** statement, and then run the program, will the pointer gets deleted?

(d) What happens if you remove the **`throw`** statement from the `C` function?

(e) What happens if you remove the **`catch`** block from the `main` function?

Exercise 4 .....