# Practice Exercise #1 **[Weight: ~3% of the Course Grade]**

## Topic: Object-Oriented Design in C++

- **For this exercise, you must work in pairs. Also, the instructor is available to provide detailed hints.**
- **Include the name and ID for each group member in your file or files.**
- **You do not have to separate class headers and class implementations for this exercise. That is, you may submit your assignment as a single `practice_exercise1.cpp` file.**
- **Please submit your completed assignment before the dropbox closes on LEARN.**
- **For this exercise, do not use pointers or dynamic memory.**

## Software Application for a Small Art Auction House

We are designing a software application for a small art auction house that handles various types of artwork.

## Step1.

Before starting this step, see corresponding lecture or tutorial notes for code that shows how to setup classes and implement operator overloading. If not sure where to find the code, speak with the instructor.

Each artwork is represented as an instance of `Artwork` class. For each piece of artwork, we need to store the artist name, year it was made, and title; year it was made is stored as an `unsigned integer` while the other attributes are stored as `string` values.

Implement the corresponding class `Artwork` that includes the required data attributes, empty constructor, parametric constructor, and overloaded `operator==`. For the empty constructor, store 0 as default year.

## Step2.

Before starting this step, see corresponding lecture or tutorial notes for code that shows how to setup inheritance. If not sure where to find the code, speak with the instructor.

Once a piece of `Artwork` has been sold, it is recorded as an instance of `SoldArtwork`, which is a derived (child) class of `Artwork`. For each sold piece, we need to store the customer name, customer address, and sale amount; the sale amount is stored as a `double` value while others are `strings`.

Implement the corresponding class `SoldArtwork` that includes the required data attributes, empty constructor, parametric constructor, and overloaded `operator==`. Getters are optional. For the empty constructor, store 0 as default sale amount.

**Step3.**

Before starting this step, see the notes included at the back of this document for code that shows how to setup and use vectors. If not sure how to interpret the code, speak with the instructor.

`ArtCollection` is used to store `Artwork` and `SoldArtwork` instances. Implement the matching class `ArtCollection`, so that it includes a vector of `Artwork` instances and another vector of `SoldArtwork` instances. Do not implement explicit constructors.

For example, to declare a vector of `Artwork` instances, write `vector<Artwork> my_artwork;`

Also, implement methods "`bool ArtCollection::insert_artwork(const Artwork& artwork_info)`" and "`bool ArtCollection::sell_artwork(const SoldArtwork& artwork_info)`".

The `insert_artwork` method inserts the given artwork into the `Artwork` vector; duplicates instances are not allowed. The `sell_artwork` method finds the corresponding `Artwork` instance, removes it from the `Artwork` vector, and then adds the `SoldArtwork` instance to the matching vector. Both methods return `true` if they succeed in their operation and `false` otherwise.

To use `SoldArtwork` instance as `Artwork`, use "`static_cast<Artwork>(artwork_info)`".

**Step4.**

Implement overloaded `operator== and operator+` functions. Implement `operator==` as a member function that checks if the two instances of `ArtCollection` are the same. Also, implement `operator+` as a non-member `friend` function that combines the two collections into one and returns a new `ArtCollection` instance with all the `Artwork` and `SoldArtwork` included.

**Step5.**

Before starting this step, see corresponding lecture or tutorial notes for explanation of assertions, stubs, and drivers; also, see the Tutorial Practice #0 solutions. If not sure what is expected, speak with the instructor.

Write a test (driver) program to test your classes and demonstrate that the specified behaviour was correctly implemented. Include one or more calls for each method specified above including constructors.

Include calls for different variants, such as trying to insert a duplicate artwork into the collection, trying to sell artwork that is not there, trying to sell the same artwork twice, and so on. The driver program should be divided into functions with appropriate names, such as `test_insert_artwork()` and `test_sell_artwork()`.

Optionally, you could create separate test classes with test methods as specified above, represent the results of each test using assertions, and then run all the included tests using a method called `run()`.

# C/C++ Vectors /1



- **<vector> Arrays: [not specific to C++11]**
  - Represent dynamic arrays in contrast to fixed-sized arrays discussed so far
  - Derived from the Standard Template Library (STL) of classes; one of container types
  - Vectors can automatically change size during program execution through internal resizing
  - Like arrays, vectors must have base type and store collection of items

```
vector<double> my_vector = {3, 2, 4, 1, 5, 7};

for (int index = 0; index < my_vector.size(); ++index)
    cout << my_vector[index] << endl;

for (auto element : my_vector)
    cout << element << endl;
```

Can initialize vectors using initializer list with C++11

Can access vectors in the same way as we accessed arrays

Can use ranged for loops from C++11 as we did with arrays

- **Vector Syntax:**
  - **Declaration: vector<<type>> <identifier>**
  - **Element Access: <identifier>[<index>] <identifier>.at(<index>)**
  - **Element Insertion at the End: <identifier>.push_back(<value>)**
  - **Element Removal from the End: <identifier>.pop_back()**

```
vector<int> my_vector; // create vector instance
for (int entry = 1; entry < 6; ++entry) {
    my_vector.push_back(entry); // insert at the end
}

my_vector.pop_back(); // remove the last element

for (int index = 0; index < my_vector.size(); ++index) {
    cout << my_vector.at(index);
} // OUTPUT: 1234
```

Obtain vector size with .size()

# C/C++ Vectors /2



- **Vector Initialization:**
  - To initialize a vector at construction, different syntax options are available including…

```
vector<double> my_vector(5); // initialize the first 5 elements as 0s
```

Empty Constructor

```
vector<double> my_vector2(10, 20.0); // the first 10 are all 20.0
```

Fill Constructor

```
// copy my_vector2 using vector iterator
vector<double> my_vector3(my_vector2.begin(), my_vector2.end());
```

Range Constructor

```
vector<double> my_vector4(my_vector3); // copy my_vector3
```

Copy Constructor

```
for (int index = 0; index < my_vector4.size(); ++index) {
    cout << my_vector4[index] << endl;
} // OUTPUT: 20 outputted 10 times
```

- **Vector Iterators:**
  - Iterators allow access to elements inside the vector without relying on vector indices
  - As a design pattern, iterators allow access to different containers using uniform interface
  - When a vector iterator is created, it can be incremented like an index but its value is one of the elements inside a vector

```
int my_array[5] = {-12, 14, 13, 11, 10}; // after this, can assign values
                                         // but cannot reinitialize this array

// copy my_array into my_vector using copy constructor
vector<int> my_vector(my_array,
                      my_array + sizeof(my_array) / sizeof(int));

// iterate through my_vector using corresponding iterator
for (vector<int>::iterator my_it = my_vector.begin();
                           my_it != my_vector.end(); ++ my_it) {
    cout << *my_it << endl; // access the value via my_it pointer
} // OUTPUT: my_array contents
```

# C/C++ Vectors /3



- **Can iterators be used with other container types such as <array>? [available in C++11]**
  - Yes — and they can be useful when it comes to conversion from fixed-size to dynamic arrays

```
array<int,50> my_array = {0};
my_array = {-12, 14, 13, 11, 10}; // fixed size <array> reinitialized
for (array<int, 50>::iterator my_it = my_array.begin();
                              my_it != my_array.end(); ++ my_it) {
   cout << *my_it << endl;
} // OUTPUT: my_array contents

// convert fixed-size <array> into dynamic <vector> array
vector<int> my_vector(my_array.begin(), my_array.end());

// iterate through my_vector using corresponding iterator
for (vector<int>::iterator my_it = my_vector.begin();
                           my_it != my_vector.end(); ++ my_it) {
   cout << *my_it << endl;
} // OUTPUT: my_array contents
```

Notice equivalent syntax

Notice equivalent syntax

- **Vector Capacity vs. Vector Size:**
  - The vector size is the number of elements currently inserted while the capacity is space available for more insertions
  - Vectors are automatically allocated extra space as needed when push_back() is called
  - Specific capacity can be pre-allocated for improved efficiency via vector reserve()
  - Cannot insert values using my_vector[index] until corresponding space is allocated

```
vector<int> my_vector; // capacity and size both at 0
my_vector.reserve(1000); // set capacity to 1000 elements
my_vector.resize(50); // set size to 50 elements

for (int entry = 0; entry < 50; entry++)
   my_vector[entry] = rand() % 100 + 1; // works due to resize

cout << "Vector capacity: " << my_vector.capacity() << endl;
cout << "Vector size: " << my_vector.size() << endl;
// OUTPUT: 1000 and 50
```