CS2810 OOAIA

Templates

Example Code Snippets

1. **Simple Class Template**

```cpp
// Basic class template with a single type parameter
template <class T>
class Box {
private:
    T item;
public:
    Box() {}
    void setItem(T newItem) { item = newItem; }
    T getItem() { return item; }
};

int main() {
    // Using the template with different types
    Box<int> intBox;
    intBox.setItem(123);

    Box<std::string> stringBox;
    stringBox.setItem("Hello Templates!");

    return 0;
}
```

2. **Function Template**

```
// Function template for finding maximum of two values
template <typename T>
T findMax(T a, T b) {
    // Works with any type that supports the > operator
    return (a > b) ? a : b;
}

int main() {
    // Same function works with different types
    int maxInt = findMax(10, 20);
    double maxDouble = findMax(3.14, 2.71);
    std::string maxString = findMax("apple", "banana");

    return 0;
}
```

3. **Template with Multiple Parameters**

```
// Template with two type parameters
template <class KeyType, class ValueType>
class Pair {
private:
    KeyType key;
    ValueType value;
public:
    Pair(KeyType k, ValueType v) : key(k), value(v) {}

    KeyType getKey() { return key; }
    ValueType getValue() { return value; }
};

int main() {
    // Using different type combinations
    Pair<int, std::string> student(101, "Alice");
```

```
    Pair<std::string, double> score("Math", 95.5);

    return 0;
}
```

4. **Template Specialization**

```
// Generic template
template <typename T>
class DataProcessor {
public:
    void process(T data) {
        std::cout << "Processing generic data" << std::endl;
    }
};

// Template specialization for string type
template <>
class DataProcessor<std::string> {
public:
    void process(std::string data) {
        std::cout << "Processing string data: " << data << std::endl;
    }
};

int main() {
    DataProcessor<int> intProcessor;
    intProcessor.process(42); // Uses generic version

    DataProcessor<std::string> stringProcessor;
    stringProcessor.process("Hello"); // Uses specialized version

    return 0;
}
```

5. **Template with Default Parameters**

```cpp
// Template with default type parameter
template <typename T, typename Container = std::vector<T>>
class Stack {
private:
    Container elements;
public:
    void push(const T& item) {
        elements.push_back(item);
    }

    T pop() {
        if (elements.empty()) {
            throw std::out_of_range("Stack is empty");
        }
        T last = elements.back();
        elements.pop_back();
        return last;
    }
};

int main() {
    // Using default container (vector)
    Stack<int> intStack;

    // Explicitly specifying a different container
    Stack<double, std::deque<double>> doubleStack;

    return 0;
}
```

6. **Non-Type Template Parameters**

```
// Template with a non-type parameter
template <typename T, int Size>
class FixedArray {
private:
    T data[Size];
public:
    FixedArray() {
        // Initialize array with default values
        for (int i = 0; i < Size; i++) {
            data[i] = T();
        }
    }

    T& operator[](int index) {
        if (index < 0 || index >= Size) {
            throw std::out_of_range("Index out of bounds");
        }
        return data[index];
    }

    int size() const { return Size; }
};

int main() {
    // Array size is specified at compile time
    FixedArray<int, 5> intArray;
    FixedArray<double, 10> doubleArray;

    return 0;
}
```

7. **Template with Variadic Parameters**

```cpp
// Variadic template for printing multiple arguments
template <typename T>
void print(T value) {
    std::cout << value << std::endl;
}

template <typename T, typename... Args>
void print(T first, Args... args) {
    std::cout << first << " ";
    print(args...); // Recursive call with remaining arguments
}

int main() {
    // Calling with different number of arguments
    print(42);
    print("Hello", "World");
    print(1, 2.5, "three", 'X');

    return 0;
}
```

Practice Problems

Problem 1: Basic Calculator

Create a template class Calculator<T> that performs basic arithmetic operations (addition, subtraction, multiplication, and division) on numeric types. The class should:

- Store two values of type T.
- Provide methods setValues(T a, T b), add(), subtract(), multiply(), and divide().
- Handle division by zero gracefully by throwing an exception.

Example Usage:

```
Calculator<int> calc;
calc.setValues(10, 5);
std::cout << calc.add() << std::endl;  // Output: 15
std::cout << calc.divide() << std::endl; // Output: 2
```

Problem 2: Dynamic Array Wrapper
Implement a template class Array<T> that wraps a dynamic array of type T. The class should:

- Use dynamic memory allocation (new and delete).
- Provide methods to:
    - Add elements (push_back(T value)).
    - Remove the last element (pop_back()).
    - Find an element (find(T value) -> bool).
    - Get the current size (size() const -> int).
    - Access elements with bounds checking (operator[]).
- Ensure proper memory cleanup in the destructor.

Example Usage:

```
Array<int> arr;
arr.push_back(10);
arr.push_back(20);
std::cout << arr[1] << std::endl;  // Output: 20
```

Problem 3: Pair Container
Create a template class PairContainer<K, V> that stores key-value pairs. The class should:

- Use an internal std::vector<std::pair<K, V>> to store pairs.
- Provide methods to:
    - Add a new pair (add(K key, V value)).
    - Find a value by key (find(K key) -> V).

    - Check if a key exists (contains(K key) -> bool).
- Handle missing keys gracefully by throwing an exception.

Example Usage:

```
PairContainer<std::string, int> studentGrades;
studentGrades.add("Alice", 95);
studentGrades.add("Bob", 88);
std::cout << studentGrades.find("Alice") << std::endl; // Output: 95
```

Problem 4: Function Template Specialization

Create a function template convert<FromType, ToType>(FromType value) that converts between different types. Implement specializations for:

- int → std::string (using std::to_string).
- std::string → int (using std::stoi).
- double → int (with rounding using std::round).
- char → int (returning ASCII value).
- Any other interesting conversions.

Example Usage:

```
std::cout << convert<int, std::string>(42) << std::endl;   // Output: "42"
std::cout << convert<std::string, int>("123") << std::endl; // Output: 123
std::cout << convert<double, int>(4.7) << std::endl; // Output: 5
std::cout << convert<char, int>('A') << std::endl; // Output: 65
```