CPP Module 07

In C++, a template is a programming construct that allows you to define generic types and functions that can work with different types of data. It is a powerful feature that enables code reuse and helps to write more flexible and efficient code.

Templates provide a way to define classes, functions, and methods that can take one or more template parameters, which are types, values, or other templates. A template parameter is specified within angle brackets '<>' after the template name, and can be used in the template code.

```
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

Templates are widely used in C++ programming for a variety of purposes, including generic data structures, algorithms, and libraries. They allow developers to write code that works with different types of data without having to duplicate the code for each type.

---

ex01

```
template<typename T>
void iter(T* array, int size_arr, void f(T&))
{
  for(int i = 0; i < size_arr; i++)
  {
    f(array[i]);
  }
}
```

```
#include <iostream>

template <typename T>
class Array {
public:
  // Default constructor
  Array() : size_(0), data_(nullptr) {}

  // Constructor with size parameter
  explicit Array(int size) : size_(size), data_(new T[size]) {}

  // Copy constructor
  Array(const Array& other) : size_(other.size_), data_(new T[other.size_]) {
    for (int i = 0; i < size_; ++i) {
      data_[i] = other.data_[i];
    }
  }

  // Copy assignment operator
  Array& operator=(const Array& other) {
    if (this != &other) {
      Array temp(other);
```

```
      swap(temp);
    }
    return *this;
  }

  // Move assignment operator
  Array& operator=(Array&& other) {
    swap(other);
    return *this;
  }

  // Destructor
  ~Array() {
    delete[] data_;
  }

  // Size accessor
  int size() const {
    return size_;
  }

  // Element accessors
  T& operator[](int index) {
    return data_[index];
  }

  const T& operator[](int index) const {
    return data_[index];
  }

  // Move constructor
  Array(Array&& other) : size_(other.size_), data_(other.data_) {
    other.size_ = 0;
    other.data_ = nullptr;
  }

  // Swap function
  void swap(Array& other) {
    std::swap(size_, other.size_);
    std::swap(data_, other.data_);
  }

private:
  int size_;
  T* data_;
};

int main() {
  // Test default constructor
  Array<int> emptyArray;
  std::cout << "emptyArray size: " << emptyArray.size() << std::endl; // Output: emptyArray size: 0

  // Test constructor with size parameter
  Array<int> intArray(10);
  intArray[0] = 5;
  intArray[1] = 10;
  std::cout << "intArray size: " << intArray.size() << std::endl; // Output: intArray size: 10
  std::cout << "intArray[0]: " << intArray[0] << std::endl; // Output: intArray[0]: 5
  std::cout << "intArray[1]: " << intArray[1] << std::endl; // Output: intArray[1]: 10

  // Test copy constructor and copy assignment operator
  Array<int> copiedArray(intArray);
  copiedArray[0] = 3;
  copiedArray[1] = 6;
  std::cout << "copiedArray size: " << copiedArray.size() << std::endl; // Output: copiedArray size: 10
  std::cout << "copiedArray[0]: " << copiedArray[0] << std::endl; // Output: copiedArray[0]: 3
  std::cout << "copiedArray[1]: " << copiedArray[1] << std::endl; // Output: copiedArray[1]: 6

  Array<int> assignedArray;
  assignedArray = copiedArray;
  std::cout << "assignedArray size: " << assignedArray.size() << std::endl; // Output: assignedArray size: 10
  std::cout << "assignedArray[0]: " << assignedArray[0] << std::endl; // Output: assignedArray[0: 3
```

```
    std::cout << "assignedArray[1]: " << assignedArray[1] << std::endl; // Output: assignedArray[1]: 6

    // Test move constructor and move assignment operator
    Array<int> movedArray(std::move(copiedArray));
    std::cout << "movedArray size: " << movedArray.size() << std::endl; // Output: movedArray size: 10
    std::cout << "movedArray[0]: " << movedArray[0] << std::endl; // Output: movedArray[0]: 3
    std::cout << "movedArray[1]: " << movedArray[1] << std::endl; // Output: movedArray[1]: 6

    Array<int> movedAssignedArray;
    movedAssignedArray = std::move(movedArray);
    std::cout << "movedAssignedArray size: " << movedAssignedArray.size() << std::endl; // Output: movedAssignedArray size: 10
    std::cout << "movedAssignedArray[0]: " << movedAssignedArray[0] << std::endl; // Output: movedAssignedArray[0]: 3
    std::cout << "movedAssignedArray[1]: " << movedAssignedArray[1] << std::endl; // Output: movedAssignedArray[1]: 6

    return 0;
    }
```