실습 12 - 13주차

학과 : 전자공학과 학번 : 2023104322 이름 : 현시온

- 과제는 pdf로 변환하여 제출(과제 문서 첫 줄에 학과/학번/이름 포함)
- 과제는 순서대로 작성하며, 문제와 설명을 모두 포함(형식이 맞지 않으면 감점)
- 프로그램을 작성하는 문제는 소스코드와 실행 결과를 모두 text로 붙여넣기(그림으로 포함하지 말 것)하고 코드 설명 및 결과에 대한 설명을 포함해야 함
- 문의 사항은 이메일(nize@khu.ac.kr) 또는 오픈 카톡방을 이용

1. 다음과 같은 예제 코드가 주석과 같이 동작하도록, 한 개의 Average 템플릿 함수와 한 개의 Average 함수(std::string 형)를 정의하라.

```
int main() {
   std::cout << Average(2.5, 3.3) << std::endl; // 2.9 출력
   std::cout << Average(2, 3) << std::endl; // 2 출력
   std::cout << Average(std::string("C++"), std::string("Programming")) << std::endl; // C++ Pro출력 ("C++" + " " + "Programming"의 앞의 1/2 문자열(소수점 이하 버림)
}
```

코드 및 설명:

```
#include <iostream>
#include <string>
template <typename T> //타입 매개변수 T로 템플릿 정의
T Average(const T& a, const T& b) {
    //임의의 타입 T의 const 레퍼런스 a, b를 매개변수로 하고 타입 T의 데이터를 반환하는 함수 Average 정의
    return (a + b) / 2; //평균값 반환.(인수가 double형이면 결과도 double, int형이면 결과도 int이므로 조건 만족)
}
std::string Average(const std::string& a, const std::string& b) {
    //문자열의 const 레퍼런스 a, b를 매개변수로 하고 문자열을 반환하는 오버로딩된 함수 Average 정의
    std::string c = a + " " + b; //문자열 병합
    return c.substr(0, (c.size() / 2)); //substr 메소드를 이용하여 절반 중 앞부분만 반환
}
```

2. 다음과 같은 예제 코드가 주석과 같이 동작하도록 Vector 템플릿 클래스와 << 연산자를 overloading하라. (<< 연사자는 double과 int에 대해서는 템플릿으로 char에 대해서는 overloading 하라)

```
int main() {
   Vector<int> v1(1234, 32644);
   Vector<char> v2(121, 22);
   Vector<double> v3(1.32, 2.234);
   std::cout << v1 << std::endl; // 1234, 32644 출력
```

```
    std::cout << v2 << std::endl; // 121, 22 출력
    std::cout << v3 << std::endl; // 1.32, 2.234 출력
}
```

코드 및 설명:

```
#include <iostream>
template <typename T> //타입 매개변수 T에 대해 템플릿 정의
class Vector { //템플릿 클래스 Vector 정의
    T x;
    T y; //임의의 타입 T의 객체 x,y를 멤버로 선언
public:
    Vector(T x, T y) : x(x), y(y) {} //파라미터 x, y가 멤버 x,y가 되도록 생성자 정의
    T getx() const { //외부에서 멤버 x, y의 복사본을 사용하기 위한 함수
        return x;
    }
    T gety() const {
        return y;
    }
};
template <typename T> //오버로드 연산자 함수는 템플릿 클래스 외부에서 선언되어야 하므로 따로 선언
std::ostream& operator<<(std::ostream& os, const Vector<T>& v) {
    os << v.getx() << ", " << v.gety(); //<<의 오버로드 형식에서 Vector의 객체의 멤버 x, y를 출력하도록 작성
    return os;
}
std::ostream& operator<<(std::ostream& os, const Vector<char>& v) { //타입 파라미터를 char로 하는 템플릿 클래스 Vector의 경우는 따로 오버로딩시켜줌
    os << static_cast<double>(v.getx()) << ", " << static_cast<double>(v.gety()); //<<의 오버로드 형식에서 Vector의 객체의 멤버 x, y를 double로 cast시켜주어 출력하도록 작성
    return os;
}
int main() {
    Vector<int> v1(1234, 32644);
    Vector<char> v2(121, 22);
    Vector<double> v3(1.32, 2.234);
    std::cout << v1 << std::endl; // 1234, 32644 출력
    std::cout << v2 << std::endl; // 121, 22 출력
    std::cout << v3 << std::endl; // 1.32, 2.234 출력
    return 0;
}
```

3. 아래 코드의 동작을 설명하라.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Data { //스트럭쳐 Data 선언 및 정의
    int x;
```

```
    Data(int x = 0) : x(x) {} //멤버 변수 x 선언 및 생성자 정의
    ~Data() {
        std::cout << "Destr.: " << x << std::endl;
    } //소멸자 정의: 객체가 소멸될 때 Destr.: 문자열과 x 값 출력
};
std::ostream& operator << (std::ostream& os, const Data& d) {
    os << d.x;
    return os;
} //연산자 <<가 Data 타입을 후피연산자로 만났을 경우, 전피연산자와, 후피연산자가 된 Data 타입의 x로 원래 연산자 <<의 역할을 하고, 전피연산자를 반환하도록 오버로딩
template <class T> //타입 매개변수 T로 템플릿 정의
class A {
    std::shared_ptr<T> p; //임의의 타입 T의 객체를 가리키는 멤버 스마트 포인터 p 선언
public:
    A(int x = 0) : p(new T(x)) {} //생성자 정의, x를 인수로 하는 타입 T의 객체를 동적으로 생성하고 p가 이것을 가리키도록 함
    T& get() const { //p의 대상을 T의 레퍼런스로 반환하는 const 메소드 get 정의
        return *p;
    }
    A copy() { //p가 가리키는 객체의 x로 클래스 A의 객체 a 생성 후 반환하는 메소드 copy 정의
        A a(p->x);
        return a;
    }
};
int main() {
    A<Data> a1, a2(10), a3(20), a4(30); //멤버 변수 x를 각각 0, 10, 20, 30으로 하여 Data 타입의 객체를 4개 생성
    std::cout << a1.get() << std::endl; //p는 a1을 가리키고, get은 a1을 반환한다. 또한 오버로드된 <<을 만나서 a1의 멤버 변수 x인 0이 출력된다. 나머지 세 객체의 출력 결과도 이와 같은 메커니즘으로 모두 멤버 변수 x의 값을 출력한다.
    std::cout << a2.get() << std::endl;
    std::cout << a3.get() << std::endl;
    std::cout << a4.get() << std::endl;

    a1 = a2.copy(); //a1에 a2의 복사본을 할당한다. 이때 스마트 포인터의 역할에 따라 원래의 a1이 소멸되어 소멸자가 작동한다.
    a3 = a4; //a3에 a4를 직접 할당한다. 이 역시 스마트 포인터의 역할에 따라 원래의 a3이 소멸되어 소멸자가 작동한다.
    std::cout << a1.get() << std::endl;
    std::cout << a2.get() << std::endl;
    std::cout << a3.get() << std::endl;
```

```
    std::cout << a4.get() << std::endl; //위와 동일한 방식으로 멤버 변수 x의 값 (10, 10, 30, 30)을 출력한다.
}
```

//이제 모든 코드의 작동이 완료되었으므로 동적으로 할당된 메모리를 전부 해제해주어야 하는데, a3과 a4의 경우 a4가 a3에 직접 할당된 관계이므로 소멸자는 a4의 것만 하나 작동하는 한편, a1과 a2의 경우 a2의 복사본이 a1에 할당된 것이므로 둘 다 동적으로 메모리가 할당된 상태였기 때문에 소멸자가 a2의 것으로 두 번 작동한다.