Lecture 8: Operator Overloading / Copy Control

SCSC 장필식

Disclaimer

오늘 하루만에 다 못 끝낼 것 같네요 ㅠㅠ

그래서 한 번 더 날짜를 잡아야 할 것 같아요

하게 될 것

- Inheritance / Polymorphism (OOP의 핵심으로 C++에서 매우 중요한 부분)
- Templates (C++에서 매우 중요한 역활이므로 빼 놓을 수가 없다...)
- 프로젝트 중간점검 (모르겠는 거나 막히는 것이 있으면 얼마든지 물어보세요)

Operators

```
auto n = 1 + 2;
auto m = 3.0 * 6.0;
n *= 3;
n++;
bool compare = !(3 < 4);
```

Operators

```
int num;
std::cout << "Enter a number: " << std::endl;
std::cin >> num;
```

Operators

```
int num;
std::cout.operator<<("Enter a number: ")
         .operator<<(std::endl);
std::cin.operator>>(num);
```

Operator Overloading

예를 들어, 2차원 벡터를 나타내는 클래스를 만들었다고 하자.

```
class Vector2 {
public:
    Vector2(float x = 0.0f, float y = 0.0f) : x(x), y(y)
    {}

    Vector2 add(const Vector2& other) {
        return Vector2(x + other.x, y + other.y);
    }

    Vector2 subtract(const Vector2& other) {
        return Vector2(x - other.x, y - other.y);
    }

    Vector2 multiply(float c) {
        return Vector2(c * x, c * y);
    }

    float x, y;
};
```

Vector2끼리의 연산을 좀 더 간편하게 만들자

```
Vector2 a(1.0f, 2.0f);
Vector2 b(3.0f, 4.0f);

Vector2 midpoint = a.add(b).multiply(0.5f); // ugly
Vector2 midpoint = 0.5f * (a + b); // much better!
```

Vector2끼리의 연산을 좀 더 간편하게 만들자

```
class Vector2 {
public:
    Vector2& operator+=(const Vector2& rhs) {
        this->x += rhs.x;
        this->y += rhs.y;
        return *this;
    }
    friend Vector2 operator+(Vector2 lhs, const Vector2& rhs) {
        lhs += rhs;
        return lhs;
    }
    ...
```

참고: friend 키워드는 함수가 Vector2의 private / protected 멤버들을 접근할 수 있도록 해준다.

Vector2끼리의 연산을 좀 더 간편하게 만들자

```
...
Vector2& operator*=(float c) {
    this->x *= c;
    this->y *= c;
    return *this;
}
friend Vector2 operator*(Vector2 lhs, float c) {
    lhs *= c;
    return lhs;
}
...
```

STL를 통해 입출력을 할 수 있도록 만들자

```
Vector2 vec;
std::cin >> vec;
std::cout << "Vector is: " << vec << std::endl;
```

STL를 통해 입출력을 할 수 있도록 만들자

클래스 바깥에 이 함수들을 만들어 주어야 한다.

(std::ostream과 std::istream가 이 함수를 바로 접근할 수 있어야 하므로)

```
std::ostream& operator<<(std::ostream& os, const Vector2& vec) {
    os << "(" << vec.x << ", " << vec.y << ")";
    return os;
}
std::istream& operator>>(std::istream& is, Vector2& vec) {
    is >> vec.x >> vec.y;
    return is;
}
```

STL를 통해 입출력을 할 수 있도록 만들자

그리고 vec의 private member들 x, y를 접근하려면 클래스 안에 다음과 같이 선언해 주어야 한다:

```
private:
    friend std::ostream&
        operator<<(std::ostream& os, const Vector2& vec);
    friend std::istream&
        operator>>(std::istream& is, Vector2& vec);

    float x, y;
};
```

Copy Control

지금까지 당연하게 생각했던 것

```
int x = 1;
int y = 2;
y = x; // x의 값이 y에 복사된다.
```

만약에 **int**가 아니라 우리가 정의한 타입이라면?

```
Person p("Dongsu", 100);
Person q("Yeongjae", 200);
Person r(p);
q = p;

cout << p.name << ", " << p.score << endl;
cout << q.name << ", " << q.score << endl;
cout << r.name << ", " << r.score << endl;
```

Copy Control

```
class Person {
public:
    // Copy constructor
    Person(const Person& other) :
        name(other.name), score(other.score) {}
    // Copy assignment operator
    Person& operator=(const Person& other) {
        name = other.name;
        score = other.score;
        return *this;
    }

private:
    std::string name;
    int score;
};
```

Copy Control (with dynamic memory)

```
class Person {
public:
    Person();
    Person(const Person& other) :
        name(other.name), score(other.score) {
        dna = new DNA();
        dna = other.dna->createClone();
    }

private:
    std::string name;
    int score;
    DNA* dna;
};
```

Rule of Three

```
class Person {
public:
    // destructor
    ~Person();
    // copy constructor
    Person(const Person&);
    // copy assignment operator
    Person& operator=(const Person&)
};
```

저 함수중에 하나라도 필요하다면, 반드시 3개 모두 구현해야 한다.

그런데... 이것만 구현하면 되는가?

복사는 꽤 비효율적이다

상황에 따라서는 복사가 안 좋을 수 있다.

Rule of three가 적용되어 있는 Str 클래스의 예를 보자.

```
class Str {
    ...
    friend Str operator+(Str lhs, const Str& rhs);
    ...
}

Str concat(const Str& a, const Str& b) {
    return a + b;
}

int main() {
    Str a("Hello ");
    Str b("World!");

    Str c(a + b); // copy occurs twice
    Str d = a + b; // same as above, copy occurs twice

    Str e = concat(a, b); // copy occurs twice
}
```

Move instead of copy?

이러한 경우에 불필요한 복사를 줄일 수 있는 방법이 없을까?

C++11 등장..

rvalue reference

C++11에서 가장 오묘한 녀석**.** 필자도 이것 때문에 가끔씩 머리를 싸매고 있다**...**

rvalue reference

지금까지의 reference (lvalue reference) 하고는 또 다른 종류.

간단히 말해서, 임시적인 값에 대한 alias이다.

타입 옆에다가 두개의 `&` 를 붙여 표현한다.

```
int i = 42;
int&& rvalue = i * 2;
```

그 전에: **lvalue vs rvalue**

"lvalues persist; rvalues are ephemeral" - C++ Primer

lvalues는 끝까지 살아남고, rvalues는 금새 날아가버린다.

즉 rvalue는 다음과 같은 값에 대한 alias여야 한다:

- 곧 있으면 없어질 값
- 아무도 사용하고 있지 않은 값

예를 들어: literals (`3, 5.6f, "hello", ...`), rvalue를 리턴하는 식 (`1+2, getNumber()`), 곧 다른 타입으로 변환되는 값

그 전에: **lvalue vs rvalue**

```
// n is lvalue, 3 is rvalue
int n = 3;
// j is lvalue, i*2 is rvalue
int j = i * 2;
// k and j are both lvalues
int k = j;
// l is lvalue, getNumber() is rvalue
int l = getNumber();
// str is lvalue, "hello" is rvalue
std::string str = "hello";
```

rvalue에 대한 미학적인 고찰**(?)**

```
int i = 42;
int&& rvalue = i * 2;
```

`i * 2` 는 어떤 값일까?

실제로 프로그램 상에서 존재하는 값일까?

존재한다면 얼마나 오랫동안 존재할까?

마치 날아가려는 나비처럼...

날아가 버리려는 값을 박제하기

`i * 2`를 곧 날아가 버릴 나비라고 가정하고, 우리는 저 펄럭거리는 나비를 잡고 싶다.

```
int newValue = i * 2;
```

우리는 나비를 잡아 newValue라는 값에 집어넣었다. 이젠 원래 나비는 존재하지 않고, newValue에 나비의 값만 남아있게 된다. (즉 이 때 `i * 2`의 값이 복사된다.)

날아가 버리려는 값을 끈끈이로 잡기

```
int& lvalueRef = i * 2; // Error!
```

나비를 죽여서 박제하는 대신 이 나비를 reference라는 끈끈이로 잡으려고 한다. 하지만 컴파일러 오류로 실패하고 만다.

만약에 누군가 이 reference를 변경한다면? `i * 2`의 값을 변경한다는 게 말이 될까?

날아가 버리려는 값을 끈끈이로 잡기

```
const int& lvalueRef = i * 2; // OK!
```

이젠 된다. 끈끈이로 나비가 영원히 달아니지 못하도록 잡아놓았다.

이젠 lvalueRef 라는 lvalue reference to const를 통해서 i * 2의 값을 접근할 수 있게 되었다.

우리의 나비는 이 reference 에 영원히 갖히게 되었다. 이 나비의 "수명"은 이젠 lvalueRef의 수명으로 연장되었다.

잠자리체로 나비를 잠시 가둬두기

우리는 끈끈이보다 좀 더 살살 나비를 잡고 싶다. 나중에 다른 변수에다 bind할 수 있도록 임시적으로 잡는 방법이 필요하다.

그래서 우리는 rvalue reference라는 잠자리채를 써 나비를 잠시 가둬둔다.

```
int&& rvalueRef = i * 2;
```

잠자리체로 나비를 잠시 가둬두기

이젠 이 rvalueRef를 가지고 뭐든지 해도 좋다.

다른 잠자리체에 가둬두거나,

```
int&& rvalueRef2 = rvalueRef;
```

값으로 박제시키거나,

```
int value = rvalueRef;
```

끈끈이로 묶어둘 수 있다.

```
const int& lvalueRef = rvalueRef;
```

Rvalue reference rules

- lvalue reference에 rvalue를 bind할 수 없다.
  - 하지만 lvalue reference to const에는 bind할 수 있다.
- rvalue reference에 임시적인 값 (`"hello"` , `3+5`)을 bind할 수 있지만, rvalue reference에 lvalue는 bind할 수 없다.

```
int i = 42;
// ok: r refers to i
int &r = i;
// error: cannot bind an rvalue ref to an lvalue
int &&rr = i;
// error: i*42 is an rvalue
int &r2 = i * 42;
// ok: we can bind a ref to const to an rvalue
const int &r3 = i * 42;
// ok: bind rr2 to the result of the multiplication
int &&rr2 = i * 42;
```

Variables are lvalues

```
// rr1 is rvalue ref, 42 is rvalue
int &&rr1 = 42; // ok
// rr2 is rvalue ref, rr1 is lvalue
int &&rr2 = rr1; // error
```

Move function

rvalue를 lvalue에 직접 bind할수는 없지만,
std::move를 통해서 lvalue를 rvalue로 캐스팅 할 수 있다.

```
int &&rr1 = 42;
int &&rr3 = std::move(rr1);
```

이때 rr1는 더 이상 값을 가지고 있지 않는다. 즉 std::move 이후에 rr1의 값을 쓰면 안된다.

(마치 unique_ptr의 용법과 비슷한것 같다면... 맞는 방향으로 생각하고 있는 것이다.)

Move function

만약에 우리가 만든 타입 Person에서 std::move를 쓸 수 있으려면, move constructor와 move assignment operator라는 두 개의 함수를 추가적으로 구현해야 한다.

```
class Person {
public:
    ...
   // move constructor
    Person(Person&&);
    // move assignment operator
    Person& operator=(Person&&);
}

Person p("Dongsu", 100);
Person&& q = std::move(p);
```

Move Semantics

자, 이젠 본격적인 예시로 우리만의 String 클래스를 만든다고 하자.
이 클래스는 std::string을 흉내내어 C string (char* 타입) 을 감싸놓은 것이다.

```
class Str {
public:
    Str(const char* p);

    ~Str();
    Str(const Str& other);
    Str& operator=(const Str& other);

    // addition operators
    Str& operator+=(const Str& rhs);
    friend Str operator+(Str lhs, const Str& rhs);

private:
    // print to ostream
    friend std::ostream&
        operator<<(std::ostream& os, const Str& str);
    char* data;
};
```

예시 코드 **(string.cpp)**

Github 링크:

Rule of Five

요약해서, 다음과 같은 함수들을 구현해야 한다. (살려줘...)

```
class Person {
public:
    // destructor
    ~Person();
    // copy constructor
    Person(const Person&);
    // copy assignment operator
    Person& operator=(const Person&)
    // move constructor
    Person(Person&&);
    // move assignment operator
    Person& operator=(Person&&);
}
```

하지만 걱정말자

항상 저렇게 만들어줘야 하는게 아니다... 겁먹지 말자.

Rule of Zero!

만약에 클래스의 멤버들이:

- primitive 타입이거나,
- Rule of Five 가 적용되어 있는 타입이면
    - 특히 shared_ptr 혹은 unique_ptr로 감싸져 있는 타입이면

저 5개의 함수들을 굳이 구현해줄 필요가 없다! (컴파일러가 알아서 적절하게 생성해줌)

즉 지금까지 만든 클래스의 거의 대부분에서 저 5개의 함수가 알아서 생성되었던 것이다!

Rule of Zero

```
class Person {
public:
    // destructor
    // ~Person() = default;

    // copy constructor
    // Person(const Person&) = default;

    // copy assignment operator
    // Person& operator=(const Person&) = default;

    // move constructor
    // Person(Person&&) = default;

    // move assignment operator
    // Person& operator=(Person&&) = default;
}
```

언제 **Rule of Five**를 적용해야 할까**?**

클래스의 멤버 중 하나라도

- Dynamic memory에 대한 포인터를 직접 가지고 있는 경우,
- 혹은 Rule of Five를 구현하고 있지 않은 경우,

저 5개의 함수들을 만들어주는 것이 좋다.

Notes

- 보통 직접 메모리를 관리하는 경우, 혹은 C 라이브러리의 구조체나 허술하게 만들어진 C++ 타입을 감쌀때 구현하게 되므로, 평상시에 많이 보게 되진 않는다. (즉 이 스터디의 범위에서는 그다지 걱정하지 않아도 될 것이다...)
- SFML 라이브러리의 소스 코드를 보면 클래스들이 모두 Rule of Five으로 잘 감싸져있다는 것을 알 수 있다.

끄으읕?