

# 10. 클래스 심화

프로그래밍 기초 교육



# 오버로딩

- 한 범위 내에서 같은 이름의 함수 또는 연산자를 여러 개 정의하는 것을 오버로딩(Overloading)이라고 한다.
- 각 함수/연산자의 매개변수는 서로 달라야 하며, 호출을 할 땐 적절한 매개변수가 있는 것을 찾아서 호출한다.

function(char)

function(double)

function(int)

호출(char)

오버로딩할 수 없는 연산자

- 멤버 연산자 .
- 포인터 연산자 \*
- 범위 연산자 ::
- 조건 표현 연산자 ? :
- sizeof()
- 멤버 연산자가 아닌 []()

클래스로 구성된 객체만 오버로딩  
될 수 있다.

# 오버로딩: 할당연산자

- 정수를 더할 때는 `a += b` 처럼 할 수 있다.
- Fraction 객체끼리 더할 땐 `+=`를 사용할 수 없고, `Fraction::addTo()`를 사용해야 했다.
- Fraction에서 사용할 수 있는 `+=` 연산자를 오버로딩 해보자.

```
Fraction fr1(4, 5);  
Fraction fr2(4, 6);  
  
fr1 += fr2;  
fr1.print();
```

`fr.operator+=(fr2)` 처럼 함수 호출 방식으로 쓸 수 있다.

그냥 함수였을 때

```
void Fraction::addTo(const Fraction&  
fr2) {  
    int numer = (numerator *  
fr2.denominator) +  
        (fr2.numerator * denominator);  
    int denom = denominator *  
fr2.denominator;  
  
    numerator = numer;  
    denominator = denom;  
  
    return;  
}
```

연산자 오버로딩

```
void Fraction::operator+=(const  
Fraction& fr2) {  
    int numer = (numerator *  
fr2.denominator) +  
        (fr2.numerator * denominator);  
    int denom = denominator *  
fr2.denominator;  
  
    numerator = numer;  
    denominator = denom;  
}
```

# 오버로딩: 산술연산자

- 정수를 더할 때는  $c = a + b$  처럼 할 수 있다.
- Fraction 객체끼리 더할 땐 +를 사용할 수 없고, 프렌드 함수인 add()를 사용해야 했다.
- Fraction에서 사용할 수 있는 + 연산자를 오버로딩 해보자.

```
Fraction fr1(4, 5);  
Fraction fr2(4, 5);
```

```
Fraction fr3 = fr1 + fr2;  
fr3.print();
```

fr3 = operator+(fr1, fr2) 처럼 함수 호출 방식으로 쓸 수 있다.

그냥 함수였을 때

```
Fraction add(const Fraction& fr1,  
             const Fraction& fr2) {  
    int numer = (fr1.numerator *  
                 fr2.denominator) +  
                (fr2.numerator *  
                 fr1.denominator);  
    int denom = fr1.denominator *  
                fr2.denominator;  
  
    return Fraction(numer, denom);  
}
```

연산자 오버로딩

```
Fraction operator+(const Fraction& fr1,  
                   const Fraction& fr2) {  
    int numer = (fr1.numerator *  
                 fr2.denominator) +  
                (fr2.numerator *  
                 fr1.denominator);  
    int denom = fr1.denominator *  
                fr2.denominator;  
  
    return Fraction(numer, denom);  
}
```

Friend 함수로 만들어놔야 쓸 수 있다.

# 오버로딩: 대입연산자

- 정수를 대입할 때는  $a = b$  처럼 할 수 있다.
- Fraction에서 사용할 수 있는  $=$  연산자를 오버로딩 해보자.

```
Fraction& Fraction::operator=
    (const Fraction& fr) {
    numerator = fr.numerator;
    denominator =
fr.denominator;
    return *this;
}
```

```
Fraction fr1(4, 5);
Fraction fr2;
Fraction fr3;
```

```
fr3 = fr2 = fr1;
fr2.print();
fr3.print();
```

대입연산자는 오른쪽부터 연산한다.

fr3 = fr2 = fr1은 (fr3 = (fr2 = fr1))로  
계산되고,  
(fr2 = fr1)에서 Fraction을 반환하기 때문에 fr3에  
Fraction이 들어갈 수 있는 것이다.

Fraction fr2 = fr; 처럼 인스턴스화 될 때  $=$ 을  
사용하면 대입연산자가 아니라 복사 생성자가 호출된다.

# 오버로딩: 전위연산자

- 정수를 증가시킬 때는 ++a 처럼 할 수 있다.
- Fraction에서 사용할 수 있는 ++ 전위연산자를 오버로딩 해보자.

```
Fraction&
Fraction::operator++() {
    numerator += denominator;
    return *this;
}
```

Fraction에 1을 더한다.

$$\frac{b}{a} + 1 = \frac{b}{a} + \frac{a}{a} = \frac{b+a}{a}$$

```
Fraction fr1(4, 5);
```

```
(++fr).print();
```

fr3 = operator++() 처럼 함수 호출 방식으로도 쓸 수 있다.

# 오버로딩: 후위연산자

- 정수를 증가시킬 때는 `a++` 처럼 할 수 있다.
- `Fraction`에서 사용할 수 있는 `++` 후위연산자를 오버로딩 해보자.

(int)에는 별 의미는 없고, 그냥 후위연산자임을 뜻한다.

```
const Fraction
Fraction::operator++(int) {

    const Fraction saved(*this);
    numerator += denominator;

    return saved;
}
```

후위연산자에서는 피연산자가 사용되고 나서 1이 더해진다.

1. 우선 복사 생성자를 호출해 원본의 `Fraction`을 만든다.
2. 더한다.
3. 반환할 땐 복사해둔 객체(심지어 `const`이다)를 반환해서 계산되기 전 값을 사용하도록 한다.

```
Fraction fr1(4, 5);
```

```
(fr++).print();
```

# 일반형에서 객체형으로 묵시적 형 변환

- 일반적인 자료형에서 객체로 바뀔 땐 매개변수가 1개 들어가는 적절한 생성자가 호출된다.
- 이를 방지하기 위해서는 해당 생성자의 프로토타입 앞에 `explicit`을 적어준다.

`int`에서 `Fraction`이 되어야 하는 상황에서 `Fraction(4)`가 호출된다.

```
Fraction fr = 4;  
cout << "Fraction: ";  
fr.print();  
cout << endl;
```

```
explicit Fraction(int numer);
```



# 오버로딩: 형 변환

- 일반적인 자료형에서 객체로 바뀔 땐 매개변수가 1개 들어가는 적절한 생성자가 호출된다.
- 이를 방지하기 위해서는 해당 생성자의 프로토타입 앞에 `explicit`을 적어준다.

알아서 float가 반환되므로 반환형은 적지 않는다.

```
Fraction::operator float()
const {
    return ((float)numerator /
            denominator);
}
```

main()에서

```
Fraction fr(4, 5);
cout << "Fraction Value: "
      << fr << endl;
```

float로 형 변환이 되므로 0.80이 출력된다.

# 오버로딩 사용해보기

기존 Fraction 클래스에 대해 fr1, fr2 객체를 생성한 후, +=, -=, \*=, /=, ++ 를 이용한 연산자 오버로딩을 통해 나오는 결과를 출력해보자.

```
ex) fr2 += fr1;          fr3 *= fr1;  
    fr2.print();        fr3.print();
```

기존 Fraction 클래스에 대해 fr1, fr2 객체를 생성한 후, +, -, \*, / 를 이용한 연산자 오버로딩을 통해 새로운 객체를 생성한 후 결과를 출력해보자.

```
ex) fr3 = fr1 + fr2;     fr4 = fr1 * fr2;  
    fr3.print();         fr4.print();
```

# 인스턴스 멤버와 정적 멤버

- 인스턴스를 만들 때마다 만들어지는 멤버를 인스턴스(instance) 멤버라고 한다.
- 반면 정적(static) 멤버는 클래스의 모든 인스턴스가 공유한다.

아래 세 가지는 정적 멤버 변수이다.

- 명시적으로 static이라고 선언된 멤버 변수
- 열거형인(enumerated) 멤버
- 형 정의된(type defined) 멤버

User  
shared

counter

2

usr1

id

1

level

13

usr2

id

2

level

67

```
class User {  
    private:  
        static int counter;  
        int id;  
        int level;  
  
    public:  
        //  
};
```

# 정적 멤버 변수

```
#include <iostream>
using namespace std;

class User {
private:
    static int counter;
    int id;
    int level;

public:
    User();
    ~User();
    void printCounter();
};

User::User() {
    this->counter++;
    // this->id = this->counter;
}

User::~User() {
    this->counter--;
}
```

```
void User::printCounter() {
    cout << "counter: " << this->counter << endl;
    return;
}
```

정적 멤버 변수의 초기화는 전역에서 해주어야 한다.

```
int User::counter = 0;

int main() {
    User usr1;
    usr1.printCounter();

    User usr2;
    usr2.printCounter();

    return 0;
}
```

```
counter: 1
counter: 2
```

# 정적 멤버 함수

- 인스턴스보다 클래스 전체와 연관된 함수는 정적으로 선언한다.
- 인스턴스가 만들어지지 않아도 사용할 수 있으며, 정적 멤버 변수에만 접근할 수 있다.

```
class User {  
private:  
    static int counter;  
    int id;  
    int level;  
  
public:  
    User();  
    ~User();  
    static void printCounter();  
};
```

```
void User::printCounter() {  
    cout << "counter: " << counter << endl;  
    return;  
}
```

각 인스턴스에 속해 있는 것이 아니므로  
본인 인스턴스를 가리키는 this를 사용할 수 없다.

```
int User::counter = 0;
```

```
int main() {  
    User usr1;  
    usr1.printCounter();  
  
    User usr2;  
    User::printCounter();  
  
    return 0;  
}
```

인스턴스에서 불러오는 것보다  
아래와 같이 클래스 이름으로  
불러오는 것이 좋다.

```
counter: 1  
counter: 2
```

# 정적 멤버 사용해보기

---

회원의 이름, 나이, e-mail을 멤버변수로 하는 Person 클래스가 있다.

Main() 함수에서 객체를 생성할 때마다 회원 수(counter, 정적 멤버 함수)가 증가하도록 설정해준다. 회원의 정보와 회원 수를 출력해보자.

(회원 수는 static void() 함수로 출력)

# 클래스 활용: 클래스 포인터

- 클래스 역시 다른 자료형처럼 포인터로 만들 수 있으며, 동적 할당 또한 할 수 있다.

\*ptr을 통해 포인터 ptr이 가리키고 있는 변수를 사용한다.

멤버 함수에서 this 포인터를 사용할 때 -> 연산자를 사용했던 것처럼, 클래스(인스턴스) 포인터의 멤버를 사용하기 위해서 포인터 연산자 없이 ->를 사용할 수 있다.

main()

```
int* intPtr = new int;  
// 정수형 포인터와 동적 할당  
  
Fraction* frPtr = new Fraction(4, 5);  
// 클래스 포인터와  
// 인스턴스의 동적 할당
```

```
(*frPtr).store(4, 6);  
(*frPtr).print();
```

```
frPtr->store(4, 9);  
frPtr->print();
```

# 클래스 활용: 클래스 배열

- 클래스 역시 다른 자료형처럼 배열에 넣을 수 있다.

```
int iArray[5] = { 1, 2, 3, 4, 5 };  
// 정수 배열 만들기
```

```
Fraction frArray1[5];
```

배열의 모든 Fraction은 기본 생성자를 사용해 만들어진다.

```
Fraction frArray2[5] =  
    { Fraction(4), Fraction(4, 6) };
```

다른 생성자를 사용하여 초기화할 수도 있다.

```
Fraction* frArray3 = new Fraction[5];
```

동적 할당을 통해 배열을 만들 수도 있다.



# 클래스 배열 사용해보기

---

Score 클래스 배열을 생성한 후 학생 4명의 수학, 과학, 영어, 국어 성적을 차례대로 입력 받는다. 이를 토대로 평균점수를 계산한 후, 학생들의 각 과목별 성적 및 평균점수를 출력해보자.

2017.05.16. 프로그래밍 기초 (2017-1)  
with D.com

1010  
01