제8장

연산자 오버로딩,
프렌드함수
그리고 참조

학습 목표

- 기본 연산자 오버로딩
  - 단항 연산자, 이항 연산자
  - 일반 함수로. 멤버 함수로.
- 프렌드와 자동 형 변환
  - 프렌드 함수, 프렌드 클래스
  - 자동 형 변환에 대한 생성자
- 참조와 다른 오버로딩
  - << 와 >>
  - 연산자들: = , [], ++, --

연산자 오버로딩 소개

- 연산자 +, -, %, ==, 등등.
  - 실제로는 함수!

- 단순히 다른 문법으로 호출됨: `x + 7`
  - "+" 는 x 와 7을 피연산자로 가지는 2항 연산자
  - 인간이 선호하는 표현법

- 다음과 같이 생각해보자: `sum(x,7)  또는  +(x, 7)`
  - "+" 는 함수 이름
  - x, 7은 인자들
  - 함수 "+" 는 인자의 합을 리턴

연산자 오버로딩 관점

- 내장 연산자들
  - 예, +, -, = , %, ==, /, *
  - 이미 C++ 내장형으로 작동

- 이러한 연산자를 오버로드!
  - 사용자 유형에 동작하도록!
  - "Chair 형", 또는 "Money 형"을 더하기
    - 사용자 요구에 대해 적절하게
    - 사용자에게 편안한 표기법으로서

- 항상 연산자의 행동과 비슷하게 오버로드!

오버로딩 기본

- 오버로딩 연산자
  - 오버로딩 함수와 매우 유사
  - 연산자 그 자체가 함수의 이름

- 선언 예제:

```
const Money operator +(const Money& amount1,
                       const Money& amount2);
```

```
bool operator ==(const Money& amount1,
                 const Money& amount2);
```

  - Money형의 피연산자에 대한 + 와 == 오버로드
    - 효율성을 위해 constant reference 매개변수 사용
    - + 의 리턴 유형은 Money형: "Money" 객체들의 덧셈을 허락
    - == 는 Money 객체를 비교하여 동일성에 대한 true/false의 부울형을 리턴
    - 두개 모두 비 멤버 연산자

오버로드된 "+" 와 "=="

- 앞선 예제에서:
  - Note : 오버로드된 "+ " 와 " == " 는 멤버 함수가 아님
  - 정의는 기본형 덧셈이나 비교보다 더 많은 것을 포함
    - money 형 덧셈이나 비교에 대한 주제를 요구함
    - 양수/음수 값을 다뤄야만 함

- 연산자 오버로드 정의는 대부분 매우 단순함
  - 단지, 사용자 유형에 특별한 덧셈이나 비교을 수행

디스플레이 8.1 연산자 오버로딩

- Money 클래스에서 "+" 연산자의 정의

```
52  const Money operator +(const Money& amount1, const Money& amount2)
53  {
54      int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
55      int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
56      int sumAllCents = allCents1 + allCents2;
57      int absAllCents = abs(sumAllCents); //Money는 음수 값을 가질 수 있음.
58      int finalDollars = absAllCents / 100;
59      int finalCents = absAllCents % 100;

60      if (sumAllCents < 0)
61      {
62          finalDollars = -finalDollars;
63          finalCents = -finalCents;
64      }

65      return Money(finalDollars, finalCents);

66  }
```

return문이 혼동을 준다면, "팁 : 생성자는 객체를 리턴한다"를 참고하라.

디스플레이 8.1 연산자 오버로딩

- Money class에서 "==" 연산자의 정의:

```
83  bool operator ==(const Money& amount1, const Money& amount2)
84  {
85      return ((amount1.getDollars( ) == amount2.getDollars( ))
86              && (amount1.getCents( ) == amount2.getCents( )));
87  }

88  const Money operator -(const Money& amount)
89  {
90      return Money(-amount.getDollars( ), -amount.getCents( ));
91  }

92  Money::Money( ): dollars(0), cents(0)
93  {/*몸체 부분은 고의적으로 비움*/}

94  Money::Money(double amount)
95                  : dollars(dollarsPart(amount)), cents(centsPart(amount))
96  {/*몸체 부분은 고의적으로 비움*/}

97  Money::Money(int theDollars)
98                  : dollars(theDollars), cents(0)
99  {/*몸체 부분은 고의적으로 비움*/}
```

원한다면, 지금까지 논했던 것처럼 다음의 짧은 생성자 정의를 인라인 함수 정의로 만들 수 있다.

객체를 리턴하는 생성자

- 생성자는 "void" 함수? 아님.
- 특별한 목적을 가진 특별한 함수
- 리턴하는 값은 클래스 객체

- Money 형을 위한 "+" 오버로드의 리턴 구문:

```
return Money(finalDollars, finalCents);
```

  - Money 클래스의 권한을 리턴!
  - 그러므로, 생성자는 실제로 객체를 리턴!
  - “익명 객체”라 함

const 값에 의한 리턴

- 다시 "+" 연산자 오버로드 고려하자:

```
const Money operator + (const Money& amount1, const Money& amount2);
```

  - "constant object" 의 리턴? 왜?

- const가 없는 객체를 리턴한다고 고려하고 살펴보자...→

```
Money operator + (const Money& amount1, const Money& amount2);
```

  - 리턴된 객체도 Money 객체이므로, 이 객체를 가지고 작업을 할 수 있다!
  - m1+m2 표현식에 의해 리턴되는 객체로 멤버 함수를 호출할 수 있다:
    - (m1+m2).output(); // 적합, 옳음? 문제없다. 어떠한 것도 변화하지 않는다
    - (m1+m2).input(); //적합! 문제 있다! 적합하지만, 변화한다!
    - 익명 객체의 수정을 허락! 여기에서 그것을 허락하면 안 된다!
  - 그러므로 const로 리턴 객체를 정의해야 함

단항 연산자 오버로딩

- C++은 단항 연산자를 가지고 있음:
  - 하나의 피연산자를 취함으로써 정의됨
  - 예, - (음수)
    - x = -y; // x 는 y의 음수와 같도록 설정
  - 다른 단항 연산자들:
    - ++, --
- 단항 연산자들도 오버로드할 수 있음

Money에서 "-" 오버로드

- 오버로드된 "-" 함수 선언
  - 클래스 정의 외부에 위치:

    const Money operator –(const Money& amount);
  - Notice: 오직 하나의 인자
    - 오직 하나의 피연산자이기 때문 (단항)

- "-" 연산자는 두 번 오버로드된다!
  - 2개의 피연산자/인자를 위해 (2항)
  - 1개의 피연산자/인자를 위해 (단항)
  - 정의는 둘 다 존재해야만 함

실습

- dollar와 cent에 대한 class Money 에 대해 아래 연산자들을 구현
  - binary operator +, -, ==, >=, <=, >, >
  - 음수 금액에 대해서는 처리하지 않음

- 아래와 같이 동작하는 stuff 구현

```
첫번째 금액을 입력하세요: 30 120
입력한 값은 31달러 20센트입니다.
두번째 금액을 입력하세요: 8 90
입력한 값은 8달러 90센트입니다.
31.20 + 8.90 = 40.10
31.20 - 8.90 = 22.30
31.20 == 8.90은 거짓
31.20 >= 8.90은 참
31.20 <= 8.90은 거짓
31.20 > 8.90은 참
31.20 < 8.90은 거짓
```

```
Money m1, m2 ;
m1.input() ; m2.input() ;

m1.output() ; cout << “+” ; m2.output ;
cout << “= “ << (m1+m2).output ;

m1.output() ; cout << “+” ; m2.output ;
cout << “= “ << (m1-m2).output ;

m1.output() ; cout << “+” ; m2.output ;
if (m1 == m2)
    cout << “은 참 "  << endl  ;
else
    cout << “은 거짓 "  << endl  ;
….
```

멤버 함수로서 오버로딩

- 앞선 예제들: 독립된 함수
  - 클래스 외부에 정의됨
- 멤버 연산자로서 오버로드 가능
  - 멤버 함수로 고려된다.
- 연산자가 멤버함수일 때는:
  - 2개가 아닌 오직 하나의 매개변수!
  - 1개의 매개변수로서 호출 객체에 제공

멤버 함수로서의 연산자 동작

```
Money  cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

- 만약 "+" 가 멤버 연산자로서 오버로드된다면:
  - 변수/객체 cost는 호출 객체
  - 객체 tax는 1개의 인자
- 다음과 같다 : `total = cost.+(tax);`

- 클래스 정의에서 "+" 의 선언:

```
const Money operator +(const Money& amount);
```

- 오직 1개의 인자라는 것에 주목

const 함수

- 언제 함수를 const로 만들까?
  - Constant 함수는 클래스 멤버 데이터를 바꾸는 것을 허락하지 않음
  - Constant 객체는 오직 constant 멤버 함수들만 호출 가능

- 좋은 코딩 스타일:
  - 데이터를 수정하지 않는 멤버 함수라 할지라도 const로 만들어라.
- 함수 선언과 머리 앞에 const 키워드를 사용

디스플레이 8.2 멤버로서의 연산자 오버로딩(1/2)

```
public:
    Money( );
    Money(double amount);
    Money(int dollars, int cents);
    Money(int dollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    void input( );
    void output( ) const;
    const Money operator +(const Money& amount2) const;
    const Money operator -(const Money& amount2) const;
    bool operator ==(const Money& amount2) const;
    const Money operator -( ) const;
private:
    int dollars;
    int cents;

    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
};
```

디스플레이 8.2 멤버로서의 연산자 오버로딩(1/2)

```
bool Money::operator ==(const Money& secondOperand) const
{
    return ((dollars == secondOperand.dollars)
            && (cents == secondOperand.cents));
}

const Money Money::operator -( ) const
{
    return Money(-dollars, -cents);
}
```

오버로딩 연산자: 어느 방법을?

- 객체 지향 프로그래밍
  - 원칙적으로 멤버 연산자를 추천
  - OOP의 정신을 유지
- 멤버 연산자는 좀 더 효율적
  - accessor와 mutator 함수 호출이 불필요
- 적어도 하나의 중요한 단점이 있음
  - (이장 후반에)

함수 호출 () 오버로딩

- 함수 호출 연산자, ( )
  - 반드시 멤버 함수로서 오버로드되어야 함
  - 클래스의 객체를 함수처럼 사용할 수 있게 함
  - 가능한 모든 인자의 개수로 오버로드 가능
- 예제:

```
Aclass anObject;
anObject(42);
```

  - 만약 ( ) 오버로드되었다면 → 오버로드 호출

  - https://www.learncpp.com/cpp-tutorial/99-overloading-the-parenthesis-operator/#:~:text=All%20of%20the%20overloaded%20operators,takes%20two%20parameters%2C%20whereas%20operator!

다른 오버로드들

- &&, ||, 그리고 콤마 연산자
  - 부울 형에 동작하는 사전 정의 버전
  - Recall: short circuit evaluation 사용
  - 오버로딩 된다면 더 이상 짧은 순환 평가를 하지 않음
    - 대신에 완전 평가 사용
    - 기대와 다르다!
- 일반적으로 이러한 연산들은 오버로딩하여 사용하지 않음

프렌드 함수

- 비멤버 함수는 accessor와 mutator 함수를 통해 데이터 접근
  - 비멤버로서 연산자 오버로드는 호출 overhead로 매우 비효율적

- 클래스의 프렌드 함수
  - 멤버 함수가 아니지만, 멤버 함수처럼 private 멤버에 직접 접근
  - 오버헤드가 없고, 더 효율적
  - 비멤버 연산자 오버로드는 프렌드로 만들면 편리
  - 함수 선언 앞에 friend 키워드 사용하여 클래스 정의에서 명시

- 일반함수로 연산자 오버로드를 구현할 때는 가능한 프렌즈로
  - accessor/mutator 멤버 함수 호출을 피하여 효율성 향상
  - 연산자는 어찌됐든 접근 할 수 있어야 함
    - 프렌드로서 전체 접근 권한을 부여 할 뿐만 아니라

프렌드 함수의 순수성

- 프렌드가 순수하지 않다?
  - 객체지향프로그래밍 측면에서 모든 함수와 연산자는 멤버 함수가 되어야
  - 많은 전문가들이 프렌드가 OOP 기본 원칙을 파괴한다고 믿음
- 장점?
  - 가능한 연산자에 대해서만 사용
    - 자동 형변환 제공
    - 최소한의 캡슐화: 프렌드는 클래스 정의 내에
    - 효율성 향상

디스플레이 8.3 프렌즈로서의 연산자 오버로딩

```
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int dollars, int cents);
    Money(int dollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    void input( );
    void output( ) const;
    friend const Money operator +(const Money& amount1, const Money& amount2);
    friend const Money operator -(const Money& amount1, const Money& amount2);
    friend bool operator ==(const Money& amount1, const Money& amount2);
    friend const Money operator -(const Money& amount);
private:
    int dollars;
    int cents;

    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
}
```

프렌드 클래스

- 클래스들도 프렌드 가능
  - 함수와 비슷하게 클래스의 프렌드가 된다.
  - 예제:

    클래스 F가 클래스 C의 프렌드이다.
    - 클래스 F의 모든 멤버 함수는 클래스 C의 프렌드 함수
    - 그 반대로는 아님!
- 문법: friend class F
  - "인증된" 클래스(클래스 C)의 정의 내부에 사용

>> 와 << 오버로딩

- 사용자가 정의한 클래스의 객체의 입/출력에 사용 가능
  - 다른 연산자 오버로드와 유사하지만 약간의 차이 존재
- 가독성 향상

```
myObject.input() ;
myObject.output();
```

```
cout << myObject;
cin >> myObject;
```

- << 오버로딩
  - cout과 사용자정의클래스에 대한 이항연산자로 구현
    - 첫 번째 피연산자는 사전 정의된 객체 cout
      - iostream 라이브러리로부터
    - 두 번째 피연산자는 모든 자료형
  - 오버로드된 <<는 어떤 값을 리턴해야 하는가?
    - cout 객체!
    - ostream 형의 첫 번째 인자를 리턴

```
cout << "Hello";
```

```
Money amount(100);
cout << "I have " << amount << endl;
```

- >> 오버로딩도 유사

디스플레이 8.5 << 와 >>의 오버로딩 (1 of 4)

디스플레이 8.5 <<와 >>의 오버로딩

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <cmath>
4   using namespace std;

5   //U.S. 통화량에 대한 클래스
6   class Money
7   {
8   public:
9       Money( );
10      Money(double amount);
11      Money(int theDollars, int theCents);
12      Money(int theDollars);
13      double getAmount( ) const;
14      int getDollars( ) const;
15      int getCents( ) const;
16      friend const Money operator +(const Money& amount1,
                                      const Money& amount2);
17      friend const Money operator -(const Money& amount1,
                                      const Money& amount2);
18      friend bool operator ==(const Money& amount1,
                                const Money& amount2);
19      friend const Money operator -(const Money& amount);
20      friend ostream& operator <<(ostream& outputStream,
                                    const Money& amount);
21      friend istream& operator >>(istream& inputStream, Money& amount);
22  private:
23      //음수량은 음수 달러와 음수 센트로 표시됨.
        //음수 $4.50는 -4와 -50으로 표현.
24      int dollars, cents;

25      int dollarsPart(double amount) const;
26      int centsPart(double amount) const;
27      int round(double number) const;
28  };
```

디스플레이 8.5 << 와 >>의 오버로딩 (2 of 4)

```
29  int main( )
30  {
31      Money yourAmount, myAmount(10, 9);
32      cout << "Enter an amount of money: ";
33      cin >> yourAmount;
34      cout << "Your amount is " << yourAmount << endl;
35      cout << "My amount is " << myAmount << endl;
36
37      if (yourAmount == myAmount)
38          cout << "We have the same amounts.\n";
39      else
40          cout << "One of us is richer.\n";

41      Money ourAmount = yourAmount + myAmount;
42      cout << yourAmount << " + " << myAmount
43           << " equals " << ourAmount << endl;

44      Money diffAmount = yourAmount - myAmount;
45      cout << yourAmount << " - " << myAmount
46           << " equals " << diffAmount << endl;

47      return 0;
48  }
```

Sample Dialogue

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

<<가 참조를 리턴하기 때문에
이것처럼 <<를 연속으로
쓸 수 있다. 같은 방법으로
>>를 연속적으로 사용할 수 있다.

디스플레이 8.5 << 와 >>의 오버로딩 (3 of 4)

```
49  ostream& operator <<(ostream& outputStream, const Money& amount)
50  {
51      int absDollars = abs(amount.dollars);
52      int absCents = abs(amount.cents);
53      if (amount.dollars < 0 || amount.cents < 0)
54          //dollars ==0과 cents ==0에 대한 계산을 위해
55          outputStream << "$-";
56      else
57          outputStream << '$';
58      outputStream << absDollars;

59      if (absCents >= 10)
60          outputStream << '.' << absCents;
61      else
62          outputStream << '.' << '0' << absCents;

63      return outputStream;
64  }
```

main 함수에서 cout이 outputStream에 대해 플러그인되었다.

입력 알고리즘을 대신하기 위하여 가장 연습문제 3번을 보라.

참조 리턴

디스플레이 8.5 << 와 >>의 오버로딩 (4 of 4)

```
66   //iostream과 cstdlib 사용:
67   istream& operator >>(istream& inputStream, Money& amount)
68   {
69       char dollarSign;
70       inputStream >> dollarSign; //희망을 가지고
71       if (dollarSign != '$')
72       {
73           cout << "No dollar sign in Money input.\n";
74           exit(1);
75       }
76       double amountAsDouble;
77       inputStream >> amountAsDouble;
78       amount.dollars = amount.dollarsPart(amountAsDouble);
79       amount.cents = amount.centsPart(amountAsDouble);

80       return inputStream;
81   }
```

main 함수에서 cin이 inputStream에 대해 틀려고인되었다.

멤버 연산자가 아니기 때문에 Money의 멤버 함수에 대해 객체를 호출하도록 기술해야 한다.

참조 리턴

할당 연산자, =

- 멤버 연산자로서 오버로딩해야 함
- 자동적으로 오버로딩
  - 디폴트 할당 연산자: 멤버간의 복사
    - 한 객체의 멤버 변수를 다른 객체의 해당 멤버 변수로 복사
- 단순한 클래스에 대해 원하는 대로 동작
  - 그러나 포인터와 함께 사용되면 → 사용자가 오버로드해야만 함!

증가와 감소

- 각 연산자는 2가지 버전이 존재
  - 전위 표현식: ++x;
  - 후위 표현식: x++;
- 오버로딩에서 구분해야 함
  - 일반적인 오버로딩 방법 → 전위
  - int형의 두 번째 매개변수 추가 → 후위
    - 컴파일러를 위한 마커 역할!
    - 후위식을 명세하도록 허락

```
class IntPair
{
public:
    IntPair::IntPair(int f, int s)  : first(f), second(s) { }
    IntPair operator++( ); //Prefix version
    IntPair operator++(int); //Postfix version
    void setFirst(int newValue);
    void setSecond(int newValue);
    int getFirst( ) const;
    int getSecond( ) const;
private:
    int first;
    int second;
};
```

```
IntPair IntPair::operator++(int ignoreMe) //postfix version
{
    int temp1 = first;
    int temp2 = second;
    first++;   second++;
    return IntPair(temp1, temp2);
}

IntPair IntPair::operator++( ) //prefix version
{
    first++;     second++;
    return IntPair(first, second);
}
```

실습

- dollar와 cent에 대한 class Money 에 대해 아래 연산자들을 구현
  - binary operator +, -, ==, >=, <=, >, >, +=, -=
  - ostream과 istream에 대해서 동작하는 <<와 >>
  - unary operator -, ++, --

- 오른쪽 같이 동작하는 stuff 구현

```
m1.input() ;
cout << “입력한 값은 “ << m1 << endl ;
m2.input() ;
cout << “입력한 값은 “ << m1 << endl ;

cout << m1 << “ + “ << m2 << “ = “
      << m1+m2 << endl ;
..
```

```
첫번째 금액을 입력하세요: 30 120
입력한 값은 $31.20
두번째 금액을 입력하세요: 8 90
입력한 값은 $8.90
31.20 + 8.90 = 40.10
31.20 - 8.90 = 22.30
31.20 == 8.90은 거짓
31.20 >= 8.90은 참
31.20 <= 8.90은 거짓
31.20 > 8.90은 참
31.20 < 8.90은 거짓
40.10 (첫번째 금액 += 두번째 금액 )
31.20 (첫번째 금액 -= 두번째 금액)
32.20 (첫번째 금액++)
31.20 (첫번째 금액 --)
```

배열 연산자 오버로딩, [ ]

- 사용자 정의 클래스에 [ ] 오버로딩 가능
  - 참조를 리턴. 연산자 [ ]는 멤버 함수로서 오버로딩!

```
class CharPair{
public:
    CharPair( ){/*Body intentionally empty*/}
    CharPair(char f, char s) : first(f), second(s) {}
    char& operator[](int index);
private:
    char first, second;
};

int main( ) {
    CharPair a;     a[1] = 'A ' ;  a[2] = 'B';
    cout << "a[1] and a[2] are:" << a[1] << a[2] << endl;

    cout << "Enter two letters (no spaces):\n";
    cin >> a[1] >> a[2];
    cout << "You entered:" << a[1] << a[2] << endl;

    return 0;
}
```

```
//Uses iostream and cstdlib:
char& CharPair::operator[](int index)
{
    if (index == 1)
        return first;
    else if (index == 2)
        return second;
    else
    {
        cout << "Illegal index value.\n";
        exit(1);
    }
}
```

요약

- C++ 내장형 연산자들을 오버로딩이 가능함
  - 사용자 정의 클래스의 객체와 동작하기 위해
- 연산자들은 실제 함수들이다.
- 프렌드 함수는 private 멤버에 직접적으로 접근
  - 프렌드 함수는 효율적이지만
  - 완벽한 accessor(getter)가 정의되어 있다면 그걸로도 충분

- 연산자들은 멤버함수로서 오버로딩이 가능
  - 첫 번째 피 연산자는 호출 객체
- 참조는 변수를 이름 짓는 한 가지 방법
- <<, >> 오버로드 가능
  - 리턴 유형은 스트림형이며, 참조형이다.

실습

- 분수와 행렬 프로그램을 아래 main으로 동작하게 변경하시오.

```
#include <iostream>
#include "Fraction.h"

int main()
{
    Fraction f1(2,3), f2(2, 5), f3 ;
    f3 = f1+f2 ;
    cout << f1 << " + " << f2 ;
    cout << " = " << f3 << endl ;
    return 0 ;
}
```

2/5 + 2/3 = 16/15

```
#include <iostream>
#include "Matrix.h"

int main()
{
    Matrix m1, m2 ;
    cout << m1 << m2 ;

    cout << "행렬 합 " << endl  ;
    cout<< m1 + m2 << endl ;

    cout << "행렬 곱 " << endl  ;
    cout<< m1 * m2 << endl ;
    return 0 ;
}
```

```
|  1   2   3 |
|  4   5   6 |
|  7   8   9 |

|  1  -1   0 |
|  0  -1   1 |
| -1   1   0 |

행렬 합
|  2   1   3 |
|  4   4   7 |
|  6   9   9 |

행렬 곱
| -2   0   2 |
| -2  -3   5 |
| -2  -6   8 |
```

실습

- 앞서 구현한 행렬 합과 곱을 다음과 같이 class Matrix를 사용하여 구현하시오.
  - main 그대로 사용
  - 파일 분리

```
|   1    2    3 |
|   4    5    6 |
|   7    8    9 |

|   1   -1    0 |
|   0   -1    1 |
|  -1    1    0 |

두 행렬의 합은
|   2    1    3 |
|   4    4    7 |
|   6    9    9 |

행렬의 곱은
|  -2    0    2 |
|  -2   -3    5 |
|  -2   -6    8 |
```

```
#include <iostream>
#include "Matrix.h"

int main()
{
    Matrix m1, m2 ;  // 자동으로 3x3 랜덤 행렬 생성.
                     // 각 요소는 -10~10 범위의 값이 되도록
    m1.print() ; m2.print() ;

    Matrix m3 = m1.add(m2) ;
    cout << "두 행렬의 합은 " << endl ;
    m3.print() ;

    m3 = m1.multi(m2) ;
    cout << "두 행렬의 곱은 " << endl ;
    m3.print() ;
    return 0 ;
}
```

과제 – D-day 계산

- class Day: 년/월/일 정보를 저장(default 생성자는 시스템 날짜)
- operator overloading
  - 단항 전위 연산자 ++와 --, 산술연산자 +와 -
    - Day d1, d2 ; d2 = d1 + 100과 같은 형태
      - +와 -의 첫 번째 인자는 Day instance, 두 번째 인자는 정수. Day 형instance.
- 실행 예제와 같이 돌아가게 구성하시오.
  - 날짜를 직접 쓰거나, 전날이나 다음날로 이동하면 현재 날짜가 변경
    - D-day가 설정되면, 설정된 D-day를 변경된 날짜에도 지속적으로 계산하게 하시오.
      - 아래 예에서 20130531의 D-300이 20120804였다가, 날짜가 20120229로 바뀌면 D-300이 20120229 기준의 D-300을 계산하여 20110505을 출력

```
[현재 날짜] 2013년 5월 31일 [D-day 없음]
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : +
[현재 날짜] 2013년 6월 1일 [D-day 없음]
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : -
[현재 날짜] 2013년 5월 31일 [D-day 없음]
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : -300
[현재 날짜] 2013년 5월 31일 [D-300] 2012년 8월 4일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : 20120229
[현재 날짜] 2012년 2월 29일 [D-300] 2011년 5월 5일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : +
[현재 날짜] 2012년 3월 1일 [D-300] 2011년 5월 6일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : 20121231
[현재 날짜] 2012년 12월 31일 [D-300] 2012년 3월 6일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : +
[현재 날짜] 2013년 1월 1일 [D-300] 2012년 3월 7일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : +100
[현재 날짜] 2013년 1월 1일 [D+100] 2013년 4월 11일
날짜 이동(년월일, (다음날)+, (전날)-), D-day 계산(+/- 날짜), 종료(Q) : Q
```