# Chapter 8. 연산자 오버로딩, 프렌드함수 그리고 참조

박 종 혁 교수
UCS Lab
(http://www.parkjonghyuk.net)

Tel: 970-6702
Email: jhpark1@snut.ac.kr

# Learning Objectives

- 기본 연산자 오버로딩
  - 단항 연산자(Unary operators)
  - 멤버 함수 오버로딩
- 프렌드와 자동 형 변환
  - 프렌드 함수, 프렌드 클래스
  - 자동 형 변환에 대한 생성자
- 다른 오버로딩 연산자와 참조
  - << 와 >>
  - 연산자: = , [], ++, --

# 연산자 오버로딩 개요

- 연산자 +, -, %, ==, etc.
  - 실제로는 함수이다!
- 단순히 다른 구문을 가지는 함수의 호출:
  x + 7
  - "+" 는 x 와 7 을 피연산자로 가지는 이항 연산자
  - 이러한 표기는 사람들에게 익숙하다
- 다음과 같이 생각해보면:
  +(x, 7)
  - "+" 는 함수의 이름
  - x와 7 은 인자
  - 함수 "+" 는 인자의 합을 리턴

# 연산자 오버로딩 관점

- Built-in 연산자
  - e.g., +, -, = , %, ==, /, *
  - C++ 기본 형에 대하여 이미 오버로딩 되어 있다
  - 기본적인 "이항 연산자" 기호
- 이들을 오버로딩할 수 있다!
  - 새로 생성한 사용자 정의 형에서 사용!
  - "Chair 형" 또는 "Money 형"에 추가
    - 사용자 정의 형에 맞게 필요한 동작을 수행
- 항상 "비슷한 동작"을 하도록 오버로딩 된다!

# 오버로딩 기본

- 오버로딩 연산자
  - 함수의 오버로딩과 매우 유사하다
  - 연산자 자체가 함수의 이름
- 선언의 예:

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

  - +를 Money형의 피연산자에 대하여 오버로딩
  - 효율성을 위하여 상수 참조 매개변수를 사용
  - 리턴값은 Money형
    - "Money“객체의 합을 수행

# 오버로딩된 "+"

- 이전의 예에서:
  - “+” 는 멤버 함수로 오버로딩 되지 않음
  - 정의는 단순한 “합”의 수행보다 더 복잡하다
    - Money 형의 특성에 맞게 고려
    - 음/양 값에 대하여 고려해야 한다
- 연산자 오버로딩의 정의는 일반적으로 단순하다
  - 사용자 정의형에 적합하게 합을 수행하도록 정의

# 디스플레이 8.1 연산자 오버로딩

- Money 클래스에 대한 "+" 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); //Money can be negative.
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  }
```

*If the* `return` *statements puzzle you, see the tip entitled* ***A Constructor Can Return an Object.***

# "==" 오버로딩

- 등가 연산자 ==
  - Money 객체의 비교 가능
  - 선언:

    bool operator ==(const Money& amount1,
    const Money& amount2);
    - 등가 판단의 결과(참/거짓)를 위한 bool 형 리턴
  - 이전의 "+"와 같이 멤버 함수가 아님

# 디스플레이 8.1 연산자 오버로딩

- Money 클래스에 대한 "==" 연산자의 정의 :

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

# 객체를 리턴하는 생성자

- 생성자는 “void” 함수?
  - Void 함수로 생각하기 쉽지만 아니다!
  - 특별한 함수
    - 특별한 특성을 가진다
    - 값을 리턴할 수 있다!
- Money 형에 대한 “+” 오버로딩에서 return 문:
  - return Money(finalDollars, finalCents);
    - Money 를 호출한다!
    - 생성자는 실제로 객체를 리턴한다!
    - “무명 객체”라고 한다

# const 값에 의한 리턴

- “+” 연산자 오버로딩의 예:

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

  – “상수형 객체”의 리턴?

  – 이유는? : 의도하지 않은 값의 변경을 방지

- 상수형 객체가 아닌 경우 →

# non-const 값에 의한 리턴

- 선언에서 const가 아닌 경우:
  Money operator +( const Money& amount1,
  const Money& amount2);
- 다음과 같은 식을 고려하면:
  m1 + m2
  - m1 과 m2는 Money 클래스의 객체
  - 리턴된 객체 또한 Money 클래스의 객체
  - 이 객체는 동작할 수 있다!
    - 멤버 함수의 호출 가능…

# non-const 객체로 할 수 있는 것

- 멤버 함수 호출 가능:
  - 식 m1+m2에 의해 리턴된 객체를 이용하여 멤버 함수 호출 가능:
    - (m1+m2).output(); //허용
      - 문제가 없다: 아무것도 바뀌지 않음
    - (m1+m2).input(); //허용!
      - 문제! //허용되지만 값이 바뀐다!
    - 무명 객체의 수정이 가능!
    - 여기에서는 허용되서는 안된다!
- 따라서, 리턴 객체를 const로 지정해야 된다

# 단항(Unary) 연산자 오버로딩

- C++ 단항 연산자를 가진다:
  - 하나의 피연산자를 갖도록 정의
  - 예 : 음수( - ) 연산자
    - x = -y;  // x가 y의 음수와 같도록 설정
  - 다른 단항 연산자:
    - ++, --
- 단항 연산자 또한 오버로딩 가능하다

# Money 클래스의 "-" 연산자 오버로딩

- 오버로딩된 "-" 연산자 함수의 선언
  - 클래스 정의의 외부에 위치:
    const Money operator –(const Money& amount);
  - 주의: 하나의 매개변수를 가진다
    - 즉 1개의 (unary) 피연산자를 가짐
- "-" 연산자는 두 번 오버로딩 된다!
  - 두 개의 피연산자/매개변수를 위해 (이항 연산자)
  - 한 개의 피연산자/매개변수를 위해 (단항 연산자)
  - 두 가지 경우의 정의가 반드시 있어야 한다!

# 오버로딩된 "-" 연산자의 정의

- 오버로딩된 "-" 함수 정의:

```
const Money operator –(const Money& amount)
{
        return Money(-amount.getDollars(),
                     -amount.getCents());
}
```

- 기본 자료형에 적용되는 "-" 단항 연산자 사용
- 무명 객체를 리턴

# 오버로딩된 "-" 단항 연산자 사용

- 예:

```
Money     amount1(10),
          amount2(6),
          amount3;
amount3 = amount1 – amount2;
```

  - 오버로딩된 이항 "-" 연산자 호출

```
amount3.output();       //Displays $4.00
amount3 = -amount1;
```

  - 오버로딩된 단항 "-" 연산자 호출

```
amount3.output()        //Displays -$10.00
```

# 멤버 함수로 오버로딩 하기

- 이전의 예: 단독(standalone) 함수
  - 클래스의 외부에 정의
- "멤버 연산자"로 오버로딩 가능
  - 다른 멤버 함수들과 동일하게 취급된다
- 연산자가 멤버함수일 경우:
  - 하나의 매개 변수, 두 개가 아니다!
  - 첫 번째 매개변수로서 객체가 호출된다

# 멤버 연산자의 동작

- Money  cost(1, 50), tax(0, 15), total;
  total = cost + tax;
  - "+" 가 멤버 연산자로 오버로딩되었을 경우:
    - cost 객체는 호출 객체
    - tax 객체는 하나의 매개변수
  - 다음과 같이 생각할 수 있다: total = cost.+(tax);
- 클래스 정의에서 "+" 연산자의 선언:
  - const Money operator +(const Money& amount);
  - 하나의 매개변수를 가지는 것에 주의!

# const 함수

- 언제 const 함수를 만드는가?
  - 상수 함수에서는 클래스 멤버 변수의 변경을 허용하지 않는다
  - 상수 객체는 상수 멤버 함수만을 호출한다!
- 좋은 코딩 스타일:
  - 데이터를 수정하지 않는 멤버 함수는 상수(const)로 만들어야 한다
- 함수의 선언과 헤더 후미에 *const* 키워드를 사용

# 연산자 오버로딩의 방법?

- Object-Oriented-Programming
  - 원칙적으로는 멤버 연산자를 추천
  - OOP의 정신에 부합한다
- 멤버 연산자가 더욱 효율적이다
  - accessor & mutator 함수의 호출이 필요없다
  - 오버헤드가 줄어든다
- 한가지 중요한 단점이 존재한다 (후반부에 언급)

# 오버로딩 함수의 응용()

- 함수 호출 연산자 ( )
  - 반드시 멤버 함수로서 오버로딩 되어야 한다
  - 클래스의 객체를 함수와 비슷하게 사용 가능
  - 가능한 매개변수 개수에 대하여 오버로딩 가능
- 예:

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

  - 만약 ( )이 오버로딩되었으면 그것이 호출된다

# 기타 오버로딩

- &&, ||, 그리고 콤마 연산자
  - 사전 정의된 버전은 bool형에 대하여 동작
  - “short-circuit evaluation” 으로 동작
  - 오버로딩 되면 짧은 순환 평가 방식으로 동작하지 않는다
    - 대신 완전 평가 방식으로 수행
    - 예상과 다른 결과
- 일반적으로 이들 연산자는 오버로딩을 하지 않음

# 프렌드 함수

- 멤버함수가 아니다
  - 멤버 함수가 아닌 함수로 오버로딩 되는 경우:
    - accessor 와 mutator 함수를 통하여 멤버 변수에 접근
    - 매우 비효율적(호출 오버헤드)
- 프렌드 함수는 클래스의 private 멤버 변수에 직접 접근 가능
  - 오버헤드가 없고 더욱 효율적이다
- 따라서 멤버 함수가 아닌 연산자의 오버로딩은 프렌드 함수로 만드는 것이 좋다!

# 프렌드 함수

- 클래스의 프렌드 함수
  - 멤버 함수가 아님
  - private 멤버 변수에 직접 접근 가능
    - 멤버 함수와 동일하게 동작
- *friend* 키워드를 함수 선언의 앞부분에 사용
  - 클래스 정의 안에 명시!
  - 멤버 함수가 아니다!

# 프렌드 함수의 사용

- 연산자 오버로딩
  - 대부분 프렌드 함수 사용
  - 효율성 향상
  - accessor/mutator 멤버 함수의 호출을 피한다
    - 멤버 함수를 통한 간접적인 접근이 필요 없다
  - 연산자는 반드시 멤버에 접근해야 한다
- 어떠한 함수도 프렌드 함수가 될 수 있다

# 프렌드 함수의 순수성

- 프렌드 함수는 순수한가?
  - OOP의 측면에서 모든 연산자와 함수는 멤버 함수가 되어야 한다고 기술한다
  - 기본적인 OOP의 규칙을 파괴한다
- 장점?
  - 연산자에서는 큰 장점!
  - 자동 형 변환을 허용
  - 여전히 캡슐화 된다: 프렌드 함수는 클래스의 정의 내에서 선언된다
  - 효율성 향상

# 프렌드 클래스

- 클래스의 전체도 프렌드가 될 수 있다
  - 프렌드 함수와 유사하다
  - 예:
    class F is friend of class C
    - 모든 클래스 F의 멤버 함수는 C의 프렌드 함수
    - 역은 성립하지 않음
    - 우정은 주는 것이지 받는 것이 아니다
- 구문: friend class F
  - "인증"해 주는 클래스의 정의 부분에 선언

# 참조(References)

- 참조의 정의:
  - 저장 위치의 이름
  - “pointer”와 유사
- 단일 참조의 예:
  - int robert;
    int& bob = robert;
    - *Bob* 는 *robert*의 저장 위치를 참조
    - *bob* 를 통하여 *robert*의 변경 가능
- 혼란스러운가?

# 참조의 사용

- 표면상으로는 위험한 방법
- 여러 경우에서 유용하다:
- Call-by-reference
  - 매커니즘의 구현에 자주 사용된다
- 참조의 리턴
  - 연산자의 오버로딩에 사용
  - 변수의 별명(alias)를 리턴

# 참조의 리턴

- 구문:
  double& sampleFunction(double& variable);
  - double& 과 double 은 다르다
  - 함수의 선언과 헤더가 반드시 동일해야 한다
- 리턴되는 아이템은 참조를 가져야 한다
  - 해당되는 형의 변수와 유사하다
  - “x+5”와 같은 식은 가능하지 않다
    - “참조해야 할” 메모리의 위치를 가지지 않는다

# 참조 리턴의 정의

- 함수 정의의 예:

```
double& sampleFunction(double& variable)
{
    return variable;
}
```

- 잘 사용하지 않는 예
- 개념의 이해를 위하여
- 주요한 사용처:
  - 특정 연산자의 오버로딩

# >> 과 << 의 오버로딩

- 사용자 객체의 입력과 출력에 사용
  - 다른 연산자 오버로딩과 비슷하다
  - 새로운 구별
- 가독성 향상
  - 모든 연산자 오버로딩 방법과 같다
  - 예:
    cout << myObject;
    cin >> myObject;
  - 다음과 같은 방법을 대체:
    myObject.output(); …

# << 오버로딩

- 삽입 연산자 <<
  - cout 과 함께 사용
  - 이항 연산자
- 예:

  cout << "Hello";
  - 연산자는 <<
  - 첫 번째 피연산자는 사전 정의된 cout 객체
    - Iostream 라이브러리 로부터
  - 두 번째 피연산자는 문자열 상수 “Hello”

# >> 오버로딩

- >>의 피연산자
  - 클래스 형의 cin 객체
  - 사용자 정의 클래스
- Money 클래스
  - output() 멤버 함수 사용보다 >> 연산자의 사용이 더욱 세련된 방법

```
Money amount(100);
cout << "I have " << amount << endl;
                대신:
cout << "I have ";
amount.output()
```

# << 오버로딩의 리턴 값

- Money amount(100);
  cout << amount;
  - << 값을 리턴해야 한다
  - 단계를 허가한다:
    cout << "I have " << amount;
    (cout << "I have ") << amount;
    - 두 식은 동일하다
- 무엇을 리턴하는가?
  - cout 객체!
    - 첫 번째 매개변수 형을 리턴

# 디스플레이 8.5 << 과 >> 의 오버로딩 (1 of 5)

**Display 8.5** **Overloading << and >>**

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

5  //Class for amounts of money in U.S. currency
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     int dollars; //A negative amount is represented as negative dollars and
24     int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

# 디스플레이 8.5 << 과 >> 의 오버로딩 (2 of 5)

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

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;
```

# 디스플레이 8.5 << 과 >> 의 오버로딩 (3 of 5)

**Display 8.5 Overloading << and >>**

```
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   }
```

*Since << returns a reference, you can chain << like this. You can chain >> in a similar way.*

*<Definitions of other member functions are as in Display 8.1. Definitions of other overloaded operators are as in Display 8.3.>*

```
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            //accounts for dollars == 0 or cents == 0
55            outputStream << "$-";
56        else
57            outputStream << '$';
58        outputStream << absDollars;
```

*In the `main` function, `cout` is plugged in for `outputStream`.*

*For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.*

# 디스플레이 8.5 << 과 >> 의 오버로딩 (4 of 5)

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

63      return outputStream;
64  }
65
66  //Uses iostream and cstdlib:
67  istream& operator >>(istream& inputStream, Money& amount)
68  {
69      char dollarSign;
70      inputStream >> dollarSign; //hopefully
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);
```

*Returns a reference*

*In the* `main` *function,* `cin` *is plugged in for* `inputStream`.

*Since this is not a member operator, you need to specify a calling object for member functions of* `Money`.

(continued)

# 디스플레이 8.5 << 과 >> 의 오버로딩 (5 of 5)

**Display 8.5 Overloading << and >>**

```
79      amount.cents = amount.centsPart(amountAsDouble);

80      return inputStream;
81  }
```

Returns a reference

**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

# 할당 연산자 =

- 반드시 멤버 연산자로 오버로딩해야 한다
- 자동으로 오버로딩 된다
  - 기본 할당 연산자:
    - Member-wise copy
    - 한 객체의 멤버 변수들 → 대응되는 다른 객체의 멤버 변수들
- 단순 클래스에 적용
  - 포인터가 포함된 경우 → 반드시 다시 작성해야 한다!

# 증가 연산자 및 감소 연산자

- 각각의 연산자는 두가지 버전이 존재
  - 전위 표기: ++x;
  - 후위 표기: x++;
- 오버로딩 시에는 반드시 구분해줘야 한다
  - 일반적인 오버로딩 방법 → 전위
  - 두 번째 매개변수로 int 형을 추가 → 후위
    - 컴파일러를 위한 마커 역할!
    - 후위 연산에 허용된다!

# 배열 연산자 [ ]

- 사용자 정의 클래스를 위하여 [] 오버로딩 가능
  - 사용자 정의 클래스의 객체와 함께 사용된다
  - 연산자는 참조를 리턴해야 한다!
  - 멤버 함수로 정의해야 한다!

# 요약 1

- C++ built-in 연산자는 오버로딩이 가능
  - 사용자 정의 클래스의 객체와 함께 동작 가능
- 연산자는 실제로 함수와 같다
- 프렌드 함수는 클래스의 private 멤버 변수에 직접 접근이 가능
- 연산자는 멤버 함수로서 오버로딩이 가능
  - 첫 번째 피연산자가 호출 객체

# 요약 2

- 프렌드 함수는 효율적이다
  - accessor/mutator 함수의 사용을 요구하지 않는다
- 참조는 변수를 별명(alias)으로 명명한다
- <<, >>는 오버로딩이 가능하다
  - 스트림 형의 참조를 리턴

# Q&A