

# 객체지향프로그래밍 II



## Lecture 4

### 11장 연산자 오버로딩 (Part 1)

1. Introduction
2. Operator Overloading 기본사항
3. Operator Overloading 제한조건





# 1. Introduction





## 연산자 오버로딩이란?

클래스 객체에 기본형 및 사용자 정의형 연산자를 사용

- ✓ 예) **Time** 클래스 객체끼리 덧셈 연산?
- ✓ 연산자 오버로딩 (**operator overloading**) 이라고 함
- ✓ 특정 클래스의 멤버 함수 호출보다 명확
- ✓ 문맥(**context**)에 따라 적응적으로 동작

예시

- ✓ **<<** : 스트림 삽입(**stream-insertion**), 비트단위 왼쪽 시프트 (**bitwise left-shift**)
- ✓ **+** : 다양한 아이템(**integers, floats, etc.**)들 간의 덧셈 연산을 수행



## 2. Operator Overloading 기본사항



# 연산자 오버로딩 방법

## Operator overloading의 타입

- ✓ 내장 (built-in) 데이터 타입 (`int, char`) 또는 사용자 정의 데이터 타입 (classes)
- ✓ 기존 연산자를 사용자 정의 타입에 사용할 수 있도록 함

- 새로운 연산자를 만들 수는 없다!

## 연산자의 오버로딩

- ✓ 클래스의 멤버 함수 정의에 의해 생성
- ✓ 연산자 멤버 함수의 이름

- `operator` 라는 키워드 뒤에 오버로드 될 연산자의 기호를 붙임
- 예) `operator+` 는 덧셈 연산자 `+`를 의미함



## 연산자 오버로딩 방법

✓ 연산자를 클래스 객체에 적용하기 위해 연산자를 반드시 오버로딩 해야 한다.

✓ 예외

- 모든 클래스에 대해 오버로딩 없이 사용할 수 있으나, 역시 프로그래머에 의해 다시 오버로딩 될 수 있는 연산자
- 대입 연산자 (=)
  - 클래스에 속한 데이터 멤버들을 멤버 별로 복사한다.
- 주소 연산자 (&)
  - 객체의 주소 값을 반환한다.



## 오버로딩은 객체간 연산의 간결한 표현을 제공

- ✓ Overloading 하지 않았을 때

```
object2 = object2.add( object1 );
```

- ✓ Overloading 했을 때

```
object2 = object2 + object1;
```



### 3. Operator Overloading 제한조건







## 연산자 오버로딩으로도 변경 불가능한 것은?

- ✓ 연산자의 우선순위 (중간 계산 순서)

- *e.g.*,  $A * B + C$

- 연산자의 순서를 바꾸기 위해서 괄호를 사용한다.

- ✓ 연산자의 결합 순서 (left-to-right 또는 right-to-left)

- ✓ 피연산자(operand)의 개수

*e.g.*,  $\&$ 은 단항(unary) 연산자이므로, 단 하나의 피연산자만을 취함

- ✓ 내장 타입의 객체에 적용되는 연산 방법  
(*i.e.*, 정수 덧셈에 사용되는  $+$  연산자의 의미를 바꿀 수 없다.)



## 연산자 오버로딩으로도 변경 불가능한 것은?

- 새로운 연산자를 만드는 것은 불가능
- 연산자는 명확하게 오버로딩 되어야 한다.
  - ✓ `=`와 `+` 연산자가 오버로딩 되었다고 해서 `+=` 연산자가 오버로딩 된 것은 아니다.
- 연산자 `?:`는 오버로딩 될 수 없다.



# Overloading 가능한/가능하지 않은 Operator

## Operators that can be overloaded

+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

## Operators that cannot be overloaded

.	.*	::	?:
---	----	----	----

# 객체지향프로그래밍 II



## Lecture 4

### 11장 연산자 오버로딩 (Part 2)

1. Operator Functions as Class Members vs. Global Functions
2. Overloading Stream Insertion and Stream Extraction Operators
3. Overloading Unary Operators
4. Overloading Binary Operators





# 1. Operator Functions as Class Members vs. Global Functions





## 멤버 함수로서의 연산자 함수

- ✓ (피연산자1 연산자 피연산자2) 일 때, 피연산자1이 연산자가 정의된 클래스의 객체가 되어야 한다.
- ✓ 이항 연산자 (binary operator)의 왼쪽 피연산자의 인자를 얻기 위해 암시적으로 **this** 키워드를 사용한다.
- ✓ 연산자 **()**, **[]**, **->** 또는 대입 연산자(**=**)는 클래스 멤버 함수로 오버로드 되어야 한다.
- ✓ 연산자 멤버 함수는 아래의 경우에만 호출된다.
  - 이항 연산자의 왼쪽 피연산자가 그 클래스의 객체인 경우
  - 단항 연산자의 피연산자가 그 클래스의 객체인 경우



# 전역 함수로서의 연산자 함수

## ☞ 전역 함수로서의 연산자 함수

- ✓ 함수 매개변수들(즉, 모든 피연산자)을 필요로 한다.
- ✓ 연산자와 다른 클래스의 객체를 매개변수로 취할 수 있다.

## ☞ 오버로드 된 << 연산자

- ✓ `ostream &` 형의 왼쪽 피연산자

- 예시 - `cout << classObject` 에서의 객체 `cout`

- ✓ 유사하게 `>>` 또한 `istream &` 형의 왼쪽 피연산자를 취한다.
- ✓ 그러므로, `<<` 와 `>>` 모두 전역 함수이다.

- `classObject` 형을 인자로 취하는 연산자 오버로딩을 위해 C++ standard library를 바꿀 수 없기 때문



# 연산자 오버로딩과 연산의 교환 법칙

## ☞ 교환 법칙이 성립하는 연산자

- ✓ 덧셈 연산자 (+) 는 교환 법칙이 성립하기를 원함

- “operand1 + operand2” 와 “operand2 + operand1” 모두 가능하도록

- ✓ 두 개의 다른 형의 객체가 피연산자일 때,


- 클래스 객체가 왼쪽에 나타날 때 오버로드 된 연산자는 클래스의 멤버 함수이어야 함

- HugelIntClass + long int

- 다른 경우일 때 전역 오버로딩 함수가 필요하다.

- long int + HugelIntClass





## 2. Overloading Stream Insertion and Stream Extraction Operators



# 연산자 오버로딩과 stream 연산자

## ☞ << 와 >> 연산자

- ✓ 내장 자료형을 처리하기 위해 이미 오버로드 되어 있음
- ✓ 사용자 정의 클래스를 처리하기 위해 전역 **friend** 함수로서 오버로딩을 구현

## ☞ 예제 프로그램

- ✓ **Class** PhoneNumber
- ✓ cout에 의해 PhoneNumber 객체의 내용을 알맞게 출력

• 예) (123) 456-7890

# 연산자 오버로딩과 stream 연산자 예제 (PhoneNumber.h)

```
1 // Fig. 11.3: PhoneNumber.h
2 // PhoneNumber class definition
3 #ifndef PHONENUMBER_H
4 #define PHONENUMBER_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15     friend ostream &operator<<( ostream &, const PhoneNumber & );
16     friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18     string areaCode; // 3-digit area code
19     string exchange; // 3-digit exchange
20     string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

Notice function prototypes for overloaded operators **>>** and **<<** (must be global, **friend** functions)

# 연산자 오버로딩과 stream 연산자 예제 (PhoneNumber.cpp)

```
1 // Fig. 11.4: PhoneNumber.cpp
2 // overloaded stream insertion and stream extraction operators
3 // for class PhoneNumber.
4 #include <iomanip>
5 using std::setw;
6
7 #include "PhoneNumber.h"
8
9 // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
14     output << "(" << number.areaCode << ")" <<
15         << number.exchange << "-" << number.line;
16     return output; // enables cout << a << b << c;
17 } // end function operator<<
```

Allows `cout << phone;` to be interpreted as: `operator<<(cout, phone) ;`

Display formatted phone number

# 연산자 오버로딩과 stream 연산자 예제 (PhoneNumber.cpp)

```
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, PhoneNumber &number )
23 {
24     input.ignore(); // skip (
25     input >> setw( 3 ) >> number.areaCode; // input area code
26     input.ignore( 2 ); // skip ) and space
27     input >> setw( 3 ) >> number.exchange; // input exchange
28     input.ignore(); // skip dash (-)
29     input >> setw( 4 ) >> number.line; // input line
30     return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

**ignore** skips specified number of characters from input (1 by default)

Input each portion of phone number separately

# 연산자 오버로딩과 stream 연산자 예제 (driver)

```
1 // Fig. 11.5: fig11_05.cpp
2 // Demonstrating class PhoneNumber's overloaded stream insertion
3 // and stream extraction operators.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "PhoneNumber.h"
10
11 int main()
12 {
13     PhoneNumber phone; // create object phone
14
15     cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17     // cin >> phone invokes operator>> by implicitly issuing
18     // the global function call operator>>( cin, phone )
19     cin >> phone;
20
21     cout << "The phone number entered was: ";
22
23     // cout << phone invokes operator<< by implicitly issuing
24     // the global function call operator<<( cout, phone )
25     cout << phone << endl;
26     return 0;
27 } // end main
```

Testing overloaded >> and << operators to input and output a **PhoneNumber** object

# 연산자 오버로딩과 stream 연산자 예제 (실행 결과)

Enter phone number in the form (123) 456-7890:

(800) 555-1212

The phone number entered was: (800) 555-1212



### 3. Overloading Unary Operators







## 단항 연산자 (unary operator) 오버로딩

- ✓ **static**이 아닌 클래스 멤버함수로 오버로딩 가능
  - 이항 연산자 오버로딩과 달리 인수가 필요없음
- ✓ 또는 하나의 인수를 갖는 전역 함수로 오버로딩 가능
  - 인수는 클래스의 객체 또는 클래스의 객체의 참조
- ✓ **static** 멤버함수는 **static** 멤버 데이터에만 접근할 수 있음을 기억할 것



## 예제 (Section 11.10)

- ✓ **String**이 비었는지 검사하기 위해 **!**를 오버로딩 한다.
- ✓ 만약 **non-static** 멤버 함수라면 인수가 필요 없다.

```
• class String  
  {  
    public:  
    bool operator!() const;  
    ...  
  };
```

- **!s** 는 **s.operator!()**로 작성된 것처럼 처리된다.

- ✓ 만약 전역 함수라면 하나의 인수가 필요하다.

```
• bool operator!(const String & )
```

- **!s** 는 **operator!(s)**로 작성된 것처럼 처리된다.



## 4. Overloading Binary Operators





## 이항 연산자 오버로딩

- ✓ **Non-static** 멤버 함수로 구현했을 때, 하나의 인수
- ✓ 전역 함수로 구현했을 때, 두 개의 인수
  - 하나의 인수는 클래스의 객체이거나 객체의 참조여야 한다.



## 예제: += 오버로딩

- ✓ **non-static** 멤버 함수라면, 하나의 인수가 필요하다.

- **class** String  
{  
  **public**  
    **const** String & **operator**+=(**const** String & );  
  ...  
};
- **y += z** 는 **y.operator**+=( z )으로 작성된 것처럼 처리된다.

- ✓ 전역 함수라면, 두 개의 인수가 필요하다.

- **const** String & **operator**+=( String &, **const** String & );
- **y +=** 는 **operator**+=( **y**, z )으로 작성된 것처럼 처리된다.