# C++ 프로그래밍

김 형 기

hk.kim@jbnu.ac.kr

# Today

- 연산자 오버로딩

# 연산자 오버로딩

# Operator Overloading

- 연산자 오버로딩 개요
- 멤버 함수인 연산자 오버로딩
- 전역 함수인 연산자 오버로딩
- 스트림 삽입 및 추출 연산자 오버로딩
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩

이번주

다음주


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩 개요
- 멤버 함수인 연산자 오버로딩
- 전역 함수인 연산자 오버로딩
- 스트림 삽입 및 추출 연산자 오버로딩
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

중요!

- 연산자 오버로딩
  - +,=,*등등의 연산자를 유저 정의 타입(ex. 클래스)에 대해 사용할 수 있도록 하는 기능
    - player1 + player2
  - 유저 정의 타입을 기본 타입(int, float, etc)과 유사하게 동작하게 할 수 있음
  - 코드를 보다 이해하기 쉽고, 작성하기 쉽게 만듬
    - 항상 좋은 것은 아님. 직관성과 가독성을 높인다고 판단될 경우 활용
  - 대입 연산자(=)를 제외하면 자동 생성되지 않으며, 사용자가 구현해 주어야 함

- 연산자 오버로딩 기능 자체도 의미가 있지만, 지금까지 배운 모든 내용이 종합적으로 적용되므로 이해하지 못한 내용이 없는지 점검해 봅시다.


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩
  - 예) 숫자 하나를 갖는 Number 클래스를 작성하고, 클래스 객체끼리의 사칙연산 기능이 필요하다고 가정. 만일 (a+b) * (c/d)를 계산해야 한다면,
  - (전역) 함수를 사용하여 구현

```
Number result = multiply(add(a,b),divide(c,d));
```


  - 멤버 함수를 사용해 구현

```
Number result = (a.add(b)).multiply(c.divide(d));
```

# Operator Overloading

- 연산자 오버로딩
  - 예) 숫자 하나를 갖는 Number 클래스를 작성하고, 클래스 객체끼리의 사칙연산 기능이 필요하다고 가정. 만일 (a+b) * (c/d)를 계산해야 한다면,
  - (전역) 함수를 사용하여 구현

```
Number result = multiply(add(a,b),divide(c,d));
```


  - 멤버 함수를 사용해 구현

```
Number result = (a.add(b)).multiply(c.divide(d));
```

  - **연산자 오버로딩을 하면,**

```
Number result = (a+b)*(c/d);
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩, 예제 클래스

```
 1 class Point
 2 {
 3 private:
 4     int xpos;
 5     int ypos;
 6 public:
 7     Point(int x=0, int y=0)
 8         :xpos{ x }, ypos{ y }
 9     {}
10     void ShowPosition() const
11     {
12         cout << "[" << xpos << "," << ypos << "]" << endl;
13     }
14
15 };
```

# Operator Overloading

- 연산자 오버로딩, 사용 목적
  - Point는 기본 자료형이 아닌 사용자 정의 자료형(클래스)임
  - 연산자 오버로딩을 통해 연산자에 의한 객체의 동작 정의 가능

```
1  Point p1{ 10,20 };
2  Point p2{ 30,40 };
3
4  Point p3 = p1 + p2;
5
6  cout << (p1 < p2) << endl;
7  cout << (p1 == p2) << endl;
8
9  cout << p3 << endl;
10
```

*현재는 불가능하지만, 이런 식으로 사용하고 싶음*

*연산자 오버로딩을 하면 가능해짐*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩, 기본 규칙
  - 연산의 우선순위를 바꿀 수는 없음 (*는 +보다 먼저 계산)
  - 단항, 이항, 삼항 연산의 교체는 불가능
  - 기본 타입(int, float, etc)에 대한 연산자는 오버로딩 불가능
  - 새로운 연산자의 정의 불가
- 연산자는 **멤버 함수 또는 전역 함수로 오버로딩 가능**!
  - 단, [ ], ( ), ->, = 등 몇몇 연산자는 멤버 함수로만 오버로딩 가능


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩
- 전역 함수인 연산자 오버로딩
- 스트림 삽입 및 추출 연산자 오버로딩
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading as Member Function

- 이항(binary) 연산자의 멤버 함수로의 선언 (+,-,==,!=,>,<, etc)
  - 선언 형태

```
Point operator+(const Point& rhs) const;
Point operator-(const Point& rhs) const;
bool operator==(const Point& rhs) const;
bool operator<(const Point& rhs) const;
```

  - 사용 예시

```
Point p1{ 10,20 };
Point p2{ 30,40 };
Point p3 = p1 + p2; //p1.operator+(p2);
p3 = p1 - p2; //p1.operator+(p2);

if(p1 == p2) //p1.operator==(p2)
    ...
```

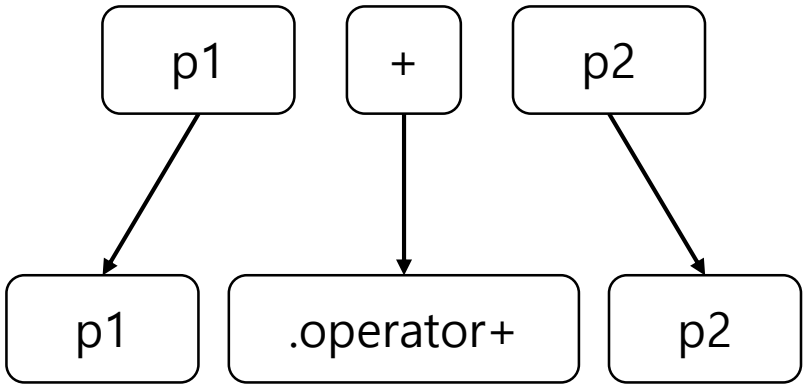

문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Binary operator, as Member Function

**중요!**

- 이항(binary) 연산자의 멤버 함수로의 선언 (+,-,==,!=,>,<, etc)
  - + 연산자의 오버로딩 동작 방식의 이해

```
1 Point operator+(const Point& rhs) const;
2
3 Point p3 = p1+p2; //p1.operator+(p2);
4     ...
```




*우리가 이러한 명령문을 작성하면, 컴파일러는 오른쪽 주석과 같이 변환한 후, 적절한 함수가 있고, 호출이 가능한지 찾습니다.*

*변환된 결과를 보면, +의 왼쪽에 있는 객체(p1)의 멤버함수인 operator+에 오른쪽 객체(p2)를 인자로 넘겨주고 있습니다.*

*즉, p1객체(Point 클래스)에 operator+ 멤버함수가 있어야 한다는 이야기입니다. 그리고 그 멤버함수는 Point객체를 인자로 받아야 합니다.*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

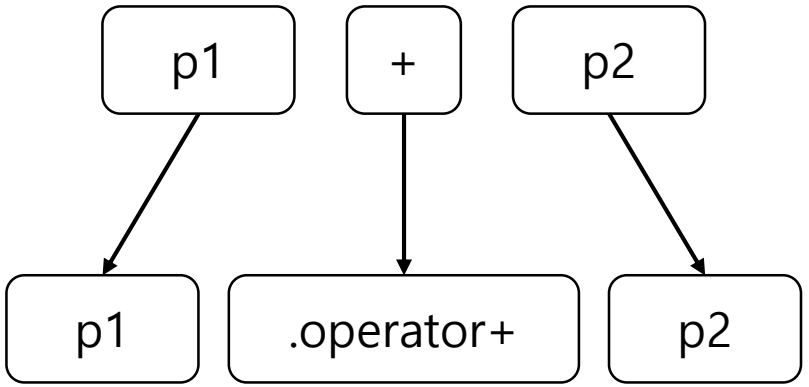
# Binary operator, as Member Function

- 이항(binary) 연산자의 멤버 함수로의 선언 (+,-,==,!=,>,<, etc)
  - + 연산자의 오버로딩 동작 방식의 이해

```
1 Point operator+(const Point& rhs) const;
2
3 Point p3 = p1+p2; //p1.operator+(p2);
4     ...
```

*p1의 값은 변하지 않으므로 const*

*p2의 값은 변하지 않으므로 const*



# Binary operator, as Member Function

- 이항(binary) 연산자의 멤버 함수로의 선언 (+,-,==,!=,>,<, etc)
  - + 연산자의 오버로딩 구현

```
1 Point operator+(const Point& rhs) const
2 {
3     return Point{ xpos + rhs.xpos, ypos + rhs.ypos };
4 }
5
6 Point p3 = p1+p2; //p1.operator+(p2);
```

*위 멤버함수가 호출되었으므로, p1과 p2의 xpos, ypos를 더한 값을 가지고있는 Point 객체가 반환되고, 그 객체는 p3에 복사됩니다.*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Binary operator, as Member Function

중요!

- 이항(binary) 연산자의 멤버 함수로의 선언 (+,-,==,!=,>,<, etc)
  - 비교 연산자의 오버로딩

```
1  bool operator==(const Point& rhs) const
2  {
3      if (xpos == rhs.xpos && ypos == rhs.ypos)
4      {
5          return true;
6      }
7      else
8      {
9          return false;
10     }
11 }
```

*p1은 변하지 않으므로 const*
*p2도 변하지 않으므로 const*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Unary operator, as Member Function

- 단항(Unary) 연산자의 멤버 함수로의 선언 (++,--,-,!)
  - 선언 형태

```
Point operator-() const;
Point operator++(); //pre-increment
Point operator++(int); //post-increment
bool operator!() const;
```

  - 사용 예시

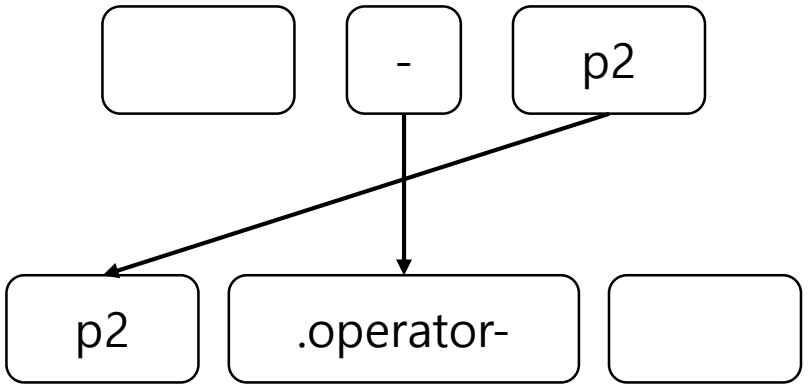
```
Point p1{10, 20};
Point p2 = -p1; //p1.operator-();
p2 = ++p1; //p1.operator++();
p2 = p1++; //p1.operator++(int);
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Unary operator, as Member Function

- 단항(Unary) 연산자의 멤버 함수로의 선언 (++,--,-,!)
  - 단항 - 연산자의 오버로딩 동작 방식의 이해

```
1 Point operator-() const;
2
3 Point p3 = -p2; //p2.operator-();
4   ...
```





문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Limitation of Member Function

중요!

- 멤버 함수로 선언 시 한계점
  - 교환 법칙이 성립하도록 구현이 불가능할 수 있음
  - 예를들어 자료형이 다른 경우, 3(int), p1(Point)를 가정해 보면,
  - p1*3은 함수 호출 가능, 3*p1은 함수 호출 불가!

```
1 Point p1{10, 20};
2 Point p2{30, 40};
3
4 Point p3 = p1 * 3; // p1.operator*(3)
5
6 Point p4 = 3 * p1; // 3.operator*(p1) ← ??
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩 : 클래스의 멤버함수로 operatorX() 라는 이름을 갖는 함수를 구현하여 연산자를 오버로딩할 수 있음. 이때 이항 연산자의 경우 우측 피연산자는 인자로 넘어옴
- 전역 함수인 연산자 오버로딩
- 스트림 삽입 및 추출 연산자 오버로딩
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Binary operator, as Global Function

- 이항(binary) 연산자의 전역 함수로의 선언 (+,-,==,!=,>,<, etc)
  - Operator 오버로딩을 전역 함수로 선언(Point::operator가 아님!)
  - Lhs도 매개변수로써 전달
    - ➢ 이러한 구현을 위해서는 함수를 friend로 선언하는 것이 일반적

```
1  Point operator+(const Point& lhs, const Point& rhs);
2  Point operator-(const Point& lhs, const Point& rhs);
3  bool operator==(const Point& lhs, const Point& rhs);
4  bool operator<(const Point& lhs, const Point& rhs);
5
6  Point p1{ 10,20 };
7  Point p2{ 30,40 };
8  Point p3 = p1 + p2; //operator+(p1,p2) or p1.operator+(p2)
9  p3 = p1 - p2; //operator-(p1,p2) or p1.operator-(p2)
10
11 if(p1 == p2) //operator==(p1,p2) or p1.operator==(p2)
12     ...
```

*우리가 이러한 명령문을 작성하면, 컴파일러는 사실 오른쪽 주석 두 개 후보 중 호출 가능한 것을 찾습니다.*

# Binary operator, as Global Function

중요!

- 이항(binary) 연산자의 전역 함수로의 선언 (+,-,==,!=,>,<, etc)
  - + 연산자의 오버로딩
  - 아래 함수는

```
1 friend Point operator+(const Point& lhs, const Point& rhs)
2 {
3     return Point{ lhs.xpos + rhs.xpos, lhs.ypos + rhs.ypos };
4 }
5
6 Point p3 = p1 + p2; //operator+(p1,p2);
```

*operator+함수는 Point 클래스의 멤버 함수가 아니기 때문에, xpos와 ypos에 쉽게 접근하기 위해 friend 선언 합니다.*

*위 전역함수가 호출되었으므로, p1과 p2의 xpos, ypos를 더한 값을 가지고있는 객체가 반환되고, 그 객체는 p3에 저장됩니다.*

*위의 operator+함수는 Point 클래스의 멤버 함수가 아님을 다시 한번 말씀 드립니다.*

# Binary operator, as Global Function

- Friend 유의사항

```
class Point
{
private:
    int xpos;
    int ypos;
public:
    Point(int x, int y)
        : xpos{ x }, ypos{ y }
    {}

    friend Point operator+(const Point& lhs, const Point& rhs)
    {
        return Point{ lhs.xpos + rhs.xpos, lhs.ypos + rhs.ypos };
    }
};
```

*operator+ 함수 코드가 class Point{...}; 안에 있다고 해서 무조건 멤버 함수가 아닙니다!*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Binary operator, as Global Function

- Friend 유의사항

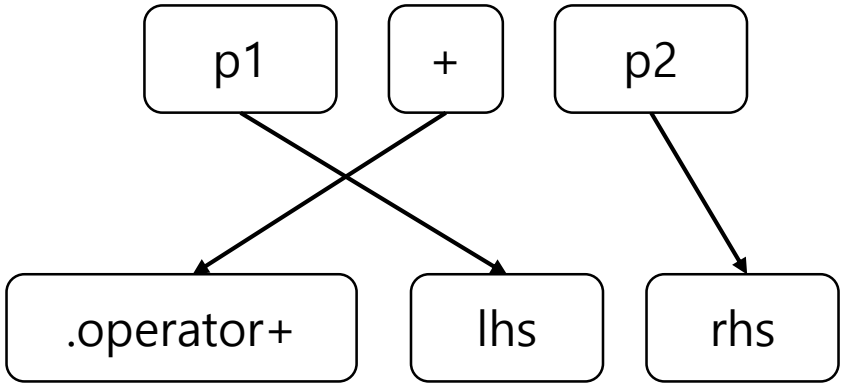
```
class Point
{
private:
    int xpos;
    int ypos;
public:
    Point(int x, int y)
        : xpos{ x }, ypos{ y }
    {}

    friend Point operator+(const Point& lhs, const Point& rhs);
};

Point operator+(const Point& lhs, const Point& rhs)
{
    return Point{ lhs.xpos + rhs.xpos, lhs.ypos + rhs.ypos };
}
```

*이전 페이지의 코드는 단지 왼쪽의 코드를 줄여 쓴 것일 뿐입니다. operator+는 전역 함수이고, class는 이 함수를 friend 선언한 것을 축약해서 쓴 것입니다.*

# Binary operator, as Global Function

- 이항(binary) 연산자의 전역 함수로의 선언 (+,-,==,!=,>,<, etc)
  - + 연산자의 오버로딩 동작 방식의 이해

```
1 friend Point operator+(const Point& lhs, const Point& rhs)
2
3 Point p3 = p1+p2; //operator+(p1,p2);
4     ...
```



# Binary operator, as Global Function

- 이항(binary) 연산자의 전역 함수로의 선언 (+,-,==,!=,>,<, etc)
  - * 연산자의 구현
    - 3*p1도 가능하도록 하기 위해 전역 함수를 추가 구현
  - 마찬가지로, 함수의 friend를 가정함
    - Friend가 아닌 경우 lhs.xpos, rhs.xpos 접근 불가

```
1 Point Point::operator*(int scale) const
2 {
3   return Point{xpos*scale, ypos*scale};
4 }
5
6 friend Point operator*(int scale, const Point& rhs)
7 {
8   return rhs*scale;
9 }
```

*p1*3 이 진입하는 멤버 함수*

*3*p1이 진입하는 전역 함수*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩 : 클래스의 멤버함수로 operatorX() 라는 이름을 갖는 함수를 구현하여 해당 클래스에 대한 연산자를 오버로딩할 수 있음. 이때 다른 피연산자는 인자로 넘어옴
- 전역 함수인 연산자 오버로딩 : 멤버 함수로 구현 시 교환법칙 문제가 발생할 수 있고, 이러한 경우 전역 함수로 오버로딩하며, 이때 friend 키워드를 사용하면 편리함
- 스트림 삽입 및 추출 연산자 오버로딩
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Stream Insertion and Extraction Overloading

- 스트림 삽입 및 추출 연산자 오버로딩
  - 구현하고자 하는 예시

```
1 Point p1{ 10,20 };
2 Point p2{ 30,40 };
3
4 p1.ShowPosition(); //[10,20]
5 p2.ShowPosition(); //[30,40]
6
7 cout << p1 << endl; //[10,20]
8 cout << p2 << endl; //[30,40]
```

← *복잡한 멤버 함수 대신,*

← *기본 자료형처럼 stream 출력 가능하도록!*

```
1 Point p3;
2 cin >> p3; //50 60
```

← *기본 자료형처럼 stream 입력도 가능하도록!*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# OStream

중요!

- cout에 대한 간단한 이해
  - <<은 cout 객체(ostream 클래스)의 오버로딩된 연산자이다

```
1  class MyOstream
2  {
3  public:
4      void operator<<(int val)
5      {
6          printf("%d", val);
7      }
8  };
9  int main()
10 {
11     cout << 123; // cout.operator<<(123);
12     cout.operator<<(123);
13     MyOstream mycout;
14     mycout << 123;
15     mycout.operator<<(123);
16     return 0;
17 }
```

*아주 간단한 버전의 cout 객체 생성을 위한 MyOstream 클래스*

*위에서 만든 클래스 객체 (mycout)을 사용한 콘솔 출력*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Stream Insertion and Extraction Overloading

중요!

- 스트림 삽입 연산자 오버로딩, 구현
  - ostream의 참조자를 반환하여 chain insertion이 가능하도록 구현해야 함
    - 참조자를 반환하지 않으면 cout << p1 << p2 와 같은 연산 불가
    - 또한 기본적으로 cout 객체는 복사가 불가

```
1 friend ostream& operator<<(ostream& os, const Point& rhs)
2 {
3     os << "[" << rhs.xpos << "," << rhs.ypos << "]";
4     return os;
5 }
```

  - 전역 함수로 선언 필요!
    - 멤버 함수로 선언하면 아래와 같이 사용해야 함...

```
1 p1 << cout; // p1.operator<<(cout);
```

# Stream Insertion and Extraction Operators Overloading

- 스트림 추출 연산자 오버로딩, 구현
  - 우측 매개변수(rhs)는 const가 아님에 유의

```
1 friend istream& operator>>(istream& is, Point& rhs)
2 {
3     int x = 0;
4     int y = 0;
5     is >> x >> y;
6     rhs = Point{ x,y };
7     return is;
8 }
```

  - 전역 함수로 선언!
    - ➢ 삽입 연산자와 마찬가지

# Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩 : 클래스의 멤버함수로 operatorX() 라는 이름을 갖는 함수를 구현하여 해당 클래스에 대한 연산자를 오버로딩할 수 있음. 이때 다른 피연산자는 인자로 넘어옴
- 전역 함수인 연산자 오버로딩 : 멤버 함수로 구현 시 교환법칙 문제가 발생할 수 있고, 이러한 경우 전역 함수로 오버로딩하며 friend 키워드를 사용하면 편리함
- 스트림 삽입 및 추출 연산자 오버로딩 : <<, >>도 연산자이며, cout/cin 객체에 대해 오버라이딩 하면 됨. Chain insertion, extraction을 위해 참조자 반환 필요
- 대입 연산자 오버로딩
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Assignment Operator Overloading

중요!

- 대입 연산자 오버로딩
  - C++는 대입 연산자를 자동 생성해줌
  - **대입 연산과 복사 생성의 구분 (아래 코드)**
  - 자동 생성된 대입 연산자는 얕은 복사를 수행

```
1 Point p1{ 10,20 };
2 Point p2{ 30,40 };
3
4 Point p3 = p1;
5 p3.ShowPosition();
6
7 Point p4;
8 p4 = p1;
9 p4.ShowPosition();
```

*대입 연산이 아닌 복사 생성!*

*여기는 대입 연산이 맞음!*

# Assignment Operator Overloading

- 대입 연산자 오버로딩, 선언
  - 기본 패턴

```
1 Type& operator=(const Type& rhs);
```

  - 참조형으로 반환하여 추가적인 복사 연산을 하지 않도록 함

*참조형으로 반환하지 않으면 어떤 일이 벌어질까?*

  - 사용 예

```
1 Point& operator=(const Point& rhs)
2
3 p4 = p1; // p4.operator=(p1);
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Assignment Operator Overloading

- 대입 연산자 오버로딩, 구현
  - 대입 연산자의 구현

*어떤 경우에 필요할까? (Hint: 멤버 변수가 포인터라면 어떤 문제가 발생할 수 있을까?)*

```
1  Point& operator=(const Point& rhs)
2  {
3      if (this == &rhs)  // 왼쪽 객체의 주소(this)가 오른쪽 객체의 주소와 같은 경우 (p1 = p1)
4          return *this;  // 왼쪽 객체의 주소(this)를 역참조하여 반환
5
6      xpos = rhs.xpos;   // 그렇지 않으면 인자로 넘어온 객체(=우변)의 멤버변수
7      ypos = rhs.ypos;   // 값을 복사해 왼쪽 객체의 멤버변수에 대입
8
9      return *this;
10 }
```

# Assignment Operator Overloading

중요!

- 대입 연산자 오버로딩, 깊은 복사
  - C++는 대입 연산자도 자동 생성해줌
  - 자동 생성된 대입 연산자는 얕은 복사를 수행
  - 포인터 타입의 멤버 변수가 존재하면, 깊은 복사를 통한 대입 연산자 직접 정의 필요!

  - "복사 생성자" 내용 복습
    - 복사 생성자는 자동 생성됨
    - 자동 생성된 복사 생성자는 얕은 복사 수행
    - **포인터 타입의 멤버 변수가 존재하는 경우, 깊은 복사의 직접 정의가 필요**


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Assignment Operator Overloading

- 대입 연산자 오버로딩, 깊은 복사
  - 예제를 위해 임시 변경된 클래스

```
1  class Array
2  {
3  private:
4      int* ptr;
5      int size;
6  public:
7      Array(int val, int size)
8        : size{size}
9      {
10         ptr = new int[size];
11         for (int i = 0; i < size; i++)
12         {
13             ptr[i] = val + i;
14         }
15     }
16     int GetSize() const
17     {
18         return size;
19     }
20     int GetValue(int index) const
21     {
22         if (index < size && index ⩾ 0)
23             return ptr[index];
24     }
25     …
26     ~Array()
27     {
28         delete[] ptr;
29     }
30 };
31
```


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Assignment Operator Overloading

중요!

- 대입 연산자 오버로딩, 깊은 복사
  - 대입 연산자의 구현

```
1  Array& operator=(const Array& rhs)
2  {
3      if (this == &rhs)
4          return *this;
5
6      delete[] ptr;
7
8      ptr = new int[rhs.size];
9
10     size = rhs.size;
11     for (int i = 0; i < size; i++)
12     {
13         ptr[i] = rhs.ptr[i];
14     }
15     return *this;
16 }
```

*Self-assignment checking이 필요함!*
*(하지 않았을 때, a = a를 작성하면 어떻게 될까?)*

*데이터를 배열로 가지고 있을 경우에는 해제(delete []) 후 재할당이 필요*

*데이터의 길이가 변할 수 있기 때문!*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Copy/Move Constructor & Overloaded Operator =

- (참고) 상속에서의 대입 연산자 오버로딩

```cpp
class Base{
    int value;
public:
    ...
    Base& operator=(const Base& rhs){
        if(this ≠ &rhs) {
            value = rhs.value;
        }
        return *this;
    }
};
```

```cpp
class Derived : public Base {
    int doubled_value;
public:
    ...
    Derived& operator=(const Derived& rhs)
        if(this ≠ &rhs) {
            Base::operator=(rhs);
            doubled_value = rhs.doubled_value;
        }
        return *this;
    }
};
```

**기본 클래스에 대한 대입 연산자를 호출하고 난 뒤, 유도 클래스의 속성에 대한 대입 연산을 처리하도록 구현*

**만일 기본 클래스 연산자를 호출하지 않는다면, value값에 대한 대입 연산은 수행되지 않음*


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Copy/Move Constructor & Overloaded Operator =

- (참고) 상속에서의 복사/이동 생성자와 오버로딩된 대입 연산자
  - 유도 클래스에서 사용자가 이를 구현하지 않은 경우,
    - 컴파일러가 자동으로 생성하며, 기본 클래스를 위한 복사/이동 생성자를 호출
  - 유도 클래스에서 사용자가 이를 구현한 경우,
    - 기본 클래스를 위한 복사/이동 생성자를 사용자가 반드시 호출해 주어야만 함!
  - 따라서, 포인터형 멤버 변수를 가지고 있는 경우, 기본 클래스의 복사/이동 생성자를 호출하는 방법에 대해 반드시 숙지해 두어야 함
    - 유도 클래스 멤버 변수에 대한 깊은 복사 고려


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩 : 클래스의 멤버함수로 operatorX() 라는 이름을 갖는 함수를 구현하여 해당 클래스에 대한 연산자를 오버로딩할 수 있음. 이때 다른 피연산자는 인자로 넘어옴
- 전역 함수인 연산자 오버로딩 : 멤버 함수로 구현 시 교환법칙 문제가 발생할 수 있고, 이러한 경우 전역 함수로 오버로딩하며 friend 키워드를 사용하면 편리함
- 스트림 삽입 및 추출 연산자 오버로딩 : <<, >>도 연산자이며, cout/cin 객체에 대해 오버라이딩 하면 됨. Chain insertion을 위해 참조자 반환
- 대입 연산자 오버로딩 : 기본 대입 연산자는 얕은 복사를 수행하기 때문에, 깊은 복사가 필요한 경우 대입 연산자 직접 오버로딩이 반드시 필요
- 첨자 연산자 오버로딩


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Subscript([ ]) operator overloading

- 첨자 연산자 오버로딩
  - [ ] 연산자
  - 멤버 함수로 오버로딩 필요
    - (복습) [ ], ( ), ->, = 와 같은 몇몇 연산자는 멤버 함수로만 오버로딩 가능
  - 경계 검사 등 기능 확장을 위해 용이하게 사용됨


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Subscript([ ]) operator overloading

- 첨자 연산자 오버로딩
  - 첨자 연산자 오버로딩 예제를 위한 클래스 정의
  - Point의 동적 할당된 배열과 그 크기를 멤버 함수로 갖는 PointArr 클래스 정의

```
1  class PointArr
2  {
3  private:
4      Point* arr;
5      int arr_len;
6
7  public:
8      PointArr(int len)
9          :arr_len{ len }
10     {
11         arr = new Point[len];
12     }
13     int get_arr_len() const { return arr_len; }
14     ~PointArr() { delete[]arr; }
15 };
```

# Subscript([ ]) operator overloading

중요!

- 첨자 연산자 오버로딩
  - 첨자 연산자 오버로딩의 구현

```
1 Point& operator[](int idx) //참조형으로 반환하는 이유?
2 {
3     if (idx < 0 || idx ⩾ arr_len)
4     {
5         cout << "Array out of bound!" << endl;
6         exit(1);
7     }
8     return arr[idx];
9 }
```

  - 참조형으로 반환해야만 arr[0] = Point{10,20}과 같이 배열 내부 데이터에 접근 가능
    - ➢ 참조형으로 반환하지 않으면 복사된 값이 반환된다는 것을 기억해야 함


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com

# Subscript([ ]) operator overloading

중요!

- 첨자 연산자 오버로딩
  - 첨자 연산자 오버로딩의 구현, 계속

```
1 Point operator[](int idx) const // const 오버로딩 하는 이유?
2 {
3     if (idx < 0 || idx ⩾ arr_len)
4     {
5         cout << "Array out of bound!" << endl;
6         exit(1);
7     }
8     return arr[idx];
9 }
```

  - const PointArr 사용에 있어서 데이터에 접근 가능해야 하기 때문

# Subscript([ ]) operator overloading

- 첨자 연산자 오버로딩에서 고려해야 할 점
  - 첨자 연산자 오버로딩에서 배열에 대한 복사 생성 허용 여부의 결정이 필요
    - ➢ 허용하지 않고 싶을 경우 아래와 같이 해상 생성자를 private 선언 또는 delete로 접근 불가능하게 함

```
private:
    Point* arr;
    int arr_len;

    PointArr(const PointArr &arr) {} //private 복사 생성자
    PointArr& operator=(const PointArr &arr) {} //private 대입 연산자
```

or

```
private:
    Point* arr;
    int arr_len;
public:
    PointArr(const PointArr &arr) = delete; //delete는 해당 함수를 구현하지 않는다는 의미
    PointArr& operator=(const PointArr &arr) = delete;
```

# (Summary)Operator Overloading

- 연산자 오버로딩 개요 : 클래스에 대한 연산자의 적용 방식을 사용자가 직접 오버로딩하여 구현할 수 있음
- 멤버 함수인 연산자 오버로딩 : 클래스의 멤버함수로 operatorX() 라는 이름을 갖는 함수를 구현하여 해당 클래스에 대한 연산자를 오버로딩할 수 있음. 이때 다른 피연산자는 인자로 넘어옴
- 전역 함수인 연산자 오버로딩 : 멤버 함수로 구현 시 교환법칙 문제가 발생할 수 있고, 이러한 경우 전역 함수로 오버로딩하며 friend 키워드를 사용하면 편리함
- 스트림 삽입 및 추출 연산자 오버로딩 : <<, >>도 연산자이며, cout/cin 객체에 대해 오버라이딩 하면 됨. Chain insertion을 위해 참조자 반환
- 대입 연산자 오버로딩 : 기본 대입 연산자는 얕은 복사를 수행하기 때문에, 깊은 복사가 필요한 경우 대입 연산자 직접 오버로딩이 반드시 필요
- 첨자 연산자 오버로딩 : 일반적으로 const 멤버와 참조 반환 함수 두개를 오버로딩하여 구현


문의사항: hk.kim@jbnu.ac.kr || diskhkme@gmail.com