# 11강. 연산자 오버로딩

## C++ 프로그래밍

jhhwang@kumoh.ac.kr

컴퓨터공학과 황준하

# 목차

# 연산자 오버로딩이란?

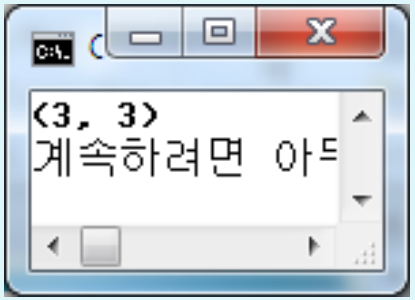
- 예 : CPoint 클래스 객체 2개를 더하는 멤버 함수 구

```
class CPoint {
private :
      int x, y;

public :
      CPoint(int a = 0, int b = 0) : x(a), y(b) { }
      CPoint Sum(const CPoint &Po) { return CPoint(x + Po.x, y + Po.y); }
      void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

void main(void)
{
      CPoint P1(1, 1);
      CPoint P2(2, 2);
      CPoint P3 = P1.Sum(P2);
      P3.Print();
}
```

P1.Sum(P2)



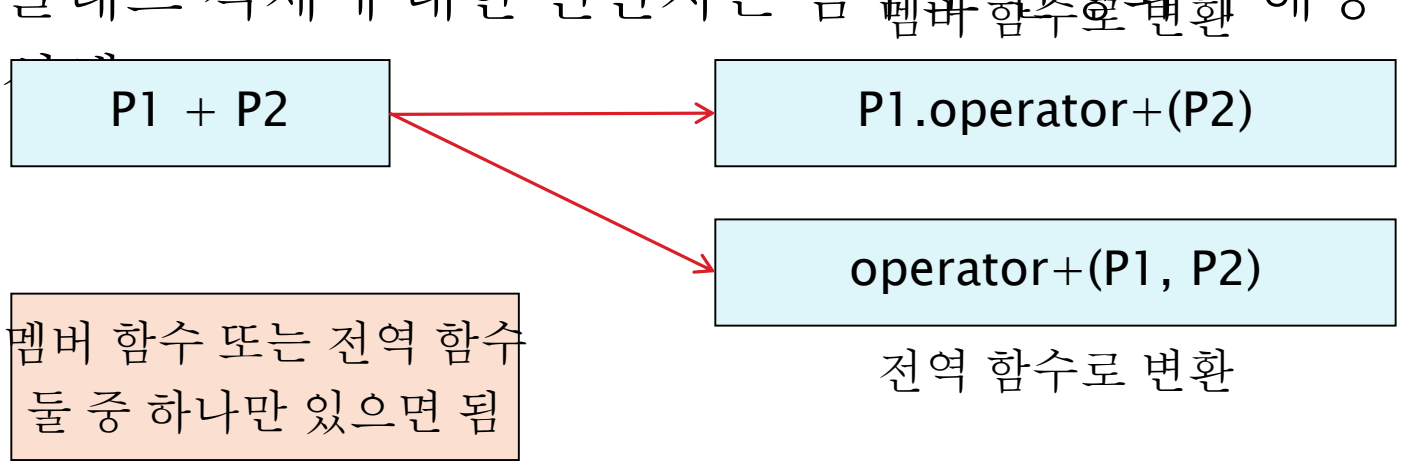

Sum 함수 : 두 객체의 x, y 좌표를 더해서 새로운 객체 반환

Sum 함수 호출

P3 = P1 + P2;
와 같이 쓸 수는 없을까?
→ 연산자 오버로딩

# 연산자 오버로딩의 원리

- 연산자 오버로딩
  - 기존의 연산자에 클래스 객체에 대한 새로운 의미를 부여하는 것
  - 일종의 함수(멤버 함수 또는 전역 함수)로 구현됨
- 연산자 오버로딩의 원리
  - 클래스 객체에 대한 연산자는 함수로 변형되어 해당 함수



# 멤버 함수에 의한 연산자 오버로딩

- P1 + P2를 멤버 함수로 구현하면?

```
class CPoint {
private :
	int x, y;

public :
	CPoint(int a = 0, int b = 0) : x(a), y(b) { }
	CPoint operator+(const CPoint &Po) { return CPoint(x + Po.x, y + Po.y); }
	void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

void main(void)
{
	CPoint P1(1, 1);
	CPoint P2(2, 2);
	CPoint P3 = P1 + P2;
	P3.Print();
}
```

함수명만 변경되고 그 외는 Sum 함수와 동일

P1.operator+(P2)

P1 + P2와 같이 사용 가능

# 전역 함수에 의한 연산자 오버로딩

- P1 + P2를 전역 함수로 구현하면?

```
CPoint operator+(const CPoint &Po1, const CPoint &Po2)
{
    return CPoint(Po1.x + Po2.x, Po1.y + Po2.y);
}

void main(void)
{
    CPoint P1(1, 1);
    CPoint P2(2, 2);
    CPoint P3 = P1 + P2;
    P3.Print();
}
```

operator+(P1, P2)

private 멤버에 대한 외부 접근

x, y에 접근할 수 있는 방법 제공 필요

1. x, y를 public 멤버로 선언
2. x, y에 접근할 수 있는 멤버 함수 제공
3. operator+ 전역함수를 friend 함수로 선언

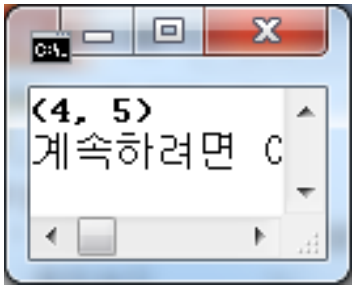
# 연산자 오버로딩 원칙

- 클래스 객체가 동반된 경우에만 가능
  - P1 + P2, P1 + 3, 3 + P1 등
- C++ 연산자 이외의 연산자 오버로딩 불가
  - P1 @ P2 (X)
- 연산자 오버로딩을 하지 않아도 사용 가능한 연산자
  - = 대입 연산자 : 멤버 단위 복사
  - & 주소 연산자 : 객체의 주소 반환
- 연산자 우선 순위 변경 불가
  - P1 + P2 * P3 : * 연산 우선 적용
- 연산자 결합 법칙 변경 불가
  - P1 + P2 + P3 : 왼쪽 +부터 적용
- 피연산자 개수 변경 불가
  - + 덧셈 연산자 : P1 + P2와 같이 두 개의 피연산자 요구

# ++ 증가 연산자 오버로딩

- 다음 main 함수가 실행될 수 있도록 하려면?

```
void main(void)
{
        CPoint P1(3, 4);
        ++P1;
        P1.Print();
}
```




P1.operator++()

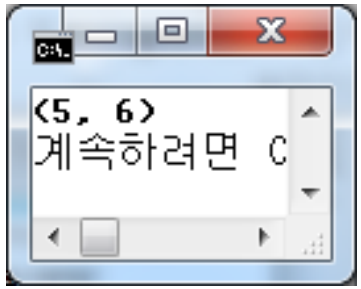
```
class CPoint {
private :
        int x, y;

public :
        CPoint(int a = 0, int b = 0) : x(a), y(b) { }
        void Print() { cout << "(" << x << ", " << y << ")" << endl; }
        void operator++(void) { ++x; ++y; }
};
```

# ++ 증가 연산자 오버로딩

- 다음과 같이 ++(++P1)이 가능하게 하려면?

```
void main(void)
{
	CPoint P1(3, 4);
	++(++P1);
	P1.Print();
}
```




Key Point : (++P1)의 결과가 다시 P1이 되어야 함

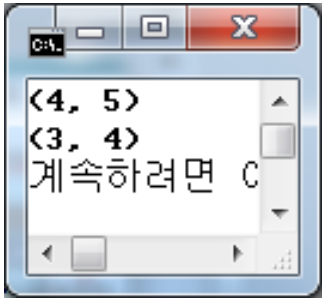
```
class CPoint {
private :
	int x, y;

public :
	CPoint(int a = 0, int b = 0) : x(a), y(b) { }
	void Print() { cout << "(" << x << ", " << y << ")" << endl; }
	CPoint &operator++(void) { ++x; ++y; return (*this); }
};
```

# ++ 증가 연산자 오버로딩

- ++ 후위 증가 연산자 오버로딩은 어떻게 구현할까?

```
void main(void)
{
        CPoint P1(3, 4);
        CPoint P2 = P1++;
        P1.Print();
        P2.Print();
}
```




P1.operator++()

No! 전위 증가 연산자와 구별이 안됨

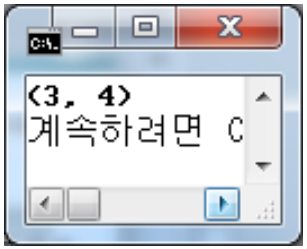
P1.operator++(0)

```
CPoint operator++(int NotUsed) {
        CPoint temp = (*this);
        ++x; ++y;
        return temp;
}
```

# ‹‹ 출력 연산자 오버로딩

- 다음과 같이 CPoint 객체를 출력할 수 있도록 하려면?

```
void main(void)
{
    CPoint P1(3, 4);
    cout << P1 << endl;
}
```




cout.operator<<(P1)

No! cout 객체의 클래스인 ostream 클래스에 operator‹‹ 멤버 함수 추가 어려움

전역 함수에 의한 연산자 오버로딩 활용

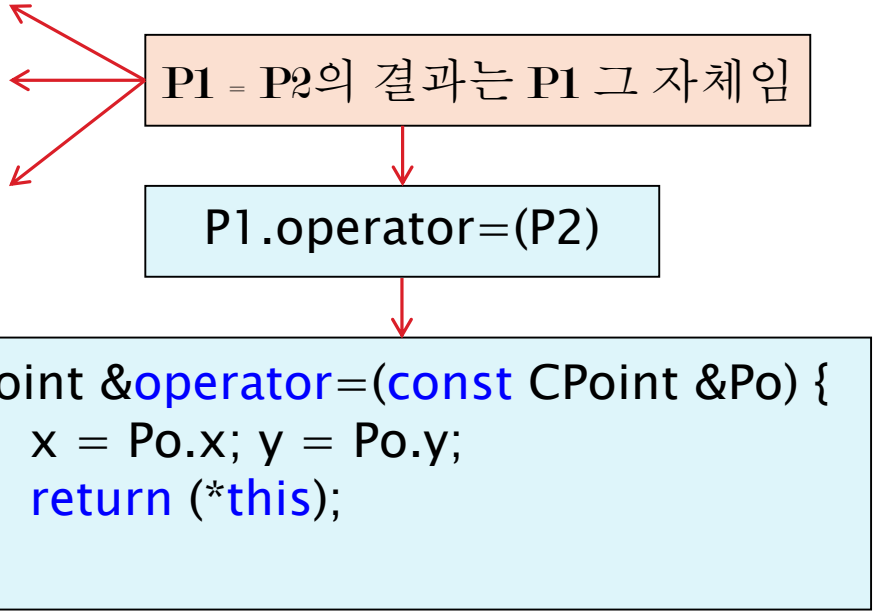
operator<<(cout, P1)

```
ostream &operator<<(ostream &out, const CPoint &Po)
{
    out << "(" << Po.x << ", " << Po.y << ")";
    return out;
}
```

주의 : private 멤버에 대한 외부 접근

# = 대입 연산자 오버로딩

- = 대입 연산자 오버로딩
  - 멤버 함수에 의한 연산자 오버로딩만 가능
- 디폴트 대입 연산자
  - 멤버 단위 복사 : 멤버 별로 대입
  - P1 = P2 가능
  - P1 = P2 = P3도 가능
  - (P1 = P2) = P3도 가능



# CString 클래스의 = 대입 연산자 오버로딩

- 다음 프로그램의 문제점은?

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

class CString {
private :
    int len;
    char *str;

public :
    CString(char *s = "Unknown") {
        len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~CString() { delete [] str; }
    void Print() { cout << str << endl; }
};
```

문자열의 길이

문자열

```
void main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = "Hello C++";

    str2 = str1;

    str1.Print();
    str2.Print();
}
```

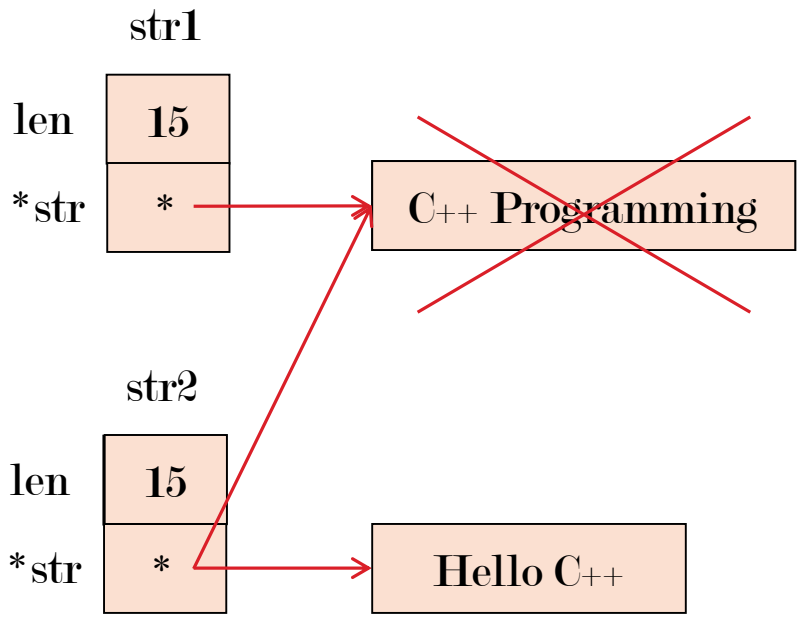
대입

생성자 : 문자열 생성

소멸자 : 문자열 메모리 해제

# CString 클래스의 = 대입 연산자 오버로딩

- 문제점 분석




```
void main(void)
{
	CString str1 = "C++ Programming";
	CString str2 = "Hello C++";

	str2 = str1;

	str1.Print();
	str2.Print();
}
```

str2 객체 소멸 → 소멸자 → delete [] str

str1 객체 소멸 → 소멸자 → delete [] str → 에러!

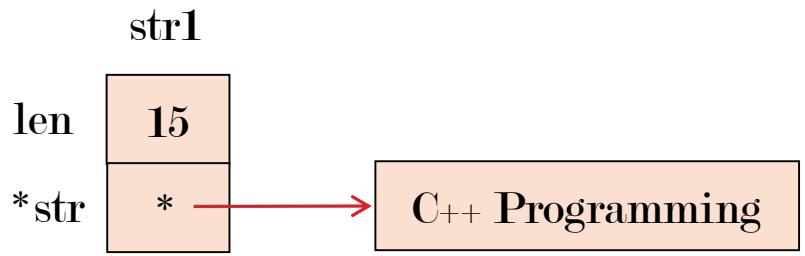
# CString 클래스의 = 대입 연산자 오버로딩

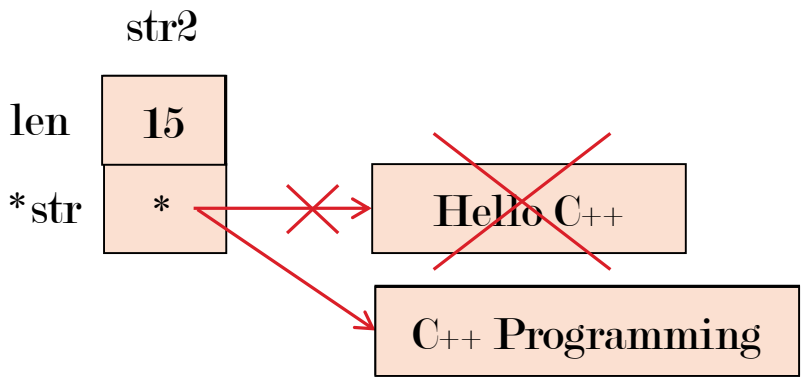
- 올바른 동작

```
void main(void)
{
        CString str1 = "C++ Programming";
        CString str2 = "Hello C++";

        str2 = str1;

        str1.Print();
        str2.Print();
}
```




1. 기존 메모리 해제
2. Len값 복사 (대입)
3. (len+1)바이트만큼 메모리 확보
4. 문자열 복사
5. 왼쪽 피연산자 그 자체 반환

# CString 클래스의 = 대입 연산자 오버로딩

- CString 클래스의 = 대입 연산자 오버로딩

```
class CString {
private :
    int len;
    char *str;

public :
    CString &operator=(const CString &S) {
        delete [] str;
        len = S.len;
        str = new char[len + 1];
        strcpy(str, S.str);
        return (*this);
    }
    // 나머지 생략
};
```

```
void main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = "Hello C++";

    str2 = str1;

    str1.Print();
    str2.Print();
}
```

1. 기존 메모리 해제
2. Len값 복사 (대입)
3. (len+1)바이트만큼 메모리 확보
4. 문자열 복사
5. 왼쪽 피연산자 그 자체 반환