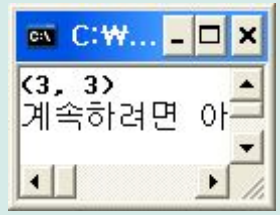
7장 연산자 오버로딩

- 연산자 오버로딩의 의미
- 덧셈(+) 연산자 오버로딩
- 연산자 오버로딩 시 고려 사항
- 이항 연산자 오버로딩
- 이항 연산자에서 피연산자의 교환 문제
- 단항 연산자 오버로딩
- 증가, 감소 단항 연산자 오버로딩
- 입출력 연산자 오버로딩을 이용한 cin, cout의 구현
- friend 함수를 사용한 입출력 연산자 오버로딩
- 대입 연산자 오버로딩
- 배열 첨자 연산자 오버로딩

1. 연산자 오버로딩의 의미

- 예 : CPoint 클래스 객체 2개를 더하는 멤버 함수 작성

```
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; }
};

int main(void)
{
    CPoint P1(1, 1);
    CPoint P2(2, 2);
    CPoint P3 = P1.Sum(P2);      // Sum 함수 호출

    P3.Print();

    return 0;
}
```

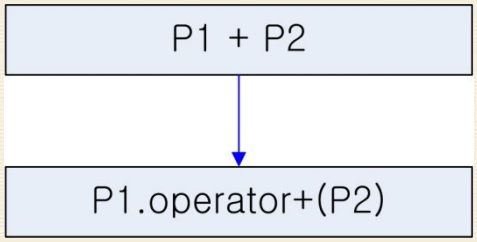


1 더하기 1은? → 1 + 1
P1 더하기 P2는? → P1 + P2?

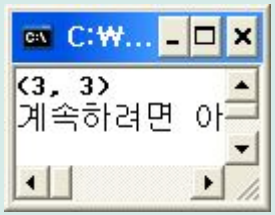
→ 연산자 오버로딩을 통해 가능!

2. 덧셈 연산자 오버로딩

- 연산자 오버로딩 방법
  - 멤버 함수에 의한 연산자 오버로딩
  - 전역 함수에 의한 연산자 오버로딩
- 멤버 함수에 의한 연산자 오버로딩 원리
  - P1 + P2의 내부 수행 방법
    - ➢ 멤버 함수 operator+ 작성



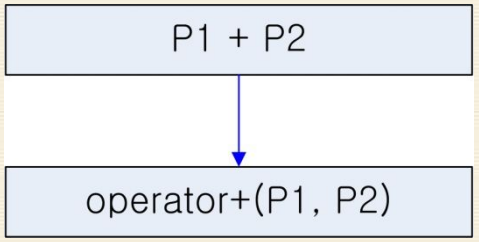

```
class CPoint {
private :
    int x, y;

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; }
};

int main(void)
{
    CPoint P1(1, 1);
    CPoint P2(2, 2);
    CPoint P3 = P1 + P2;  // + 연산자 호출

    P3.Print();
    return 0;
}
```



명시적 호출 가능 : **P1.operator+(P2);**

2. 덧셈 연산자 오버로딩

- 전역 함수에 의한 연산자 오버로딩
  - P1 + P2의 내부 수행 방법
    - 전역 함수 operator+ 작성
  - 다음 프로그램의 문제점은?




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

전역 함수에 의한 연산자 오버로딩

CPoint 클래스의 private 멤버로의 접근 → 불가능
해결 방법 : friend

2. 덧셈 연산자 오버로딩

⌗ 전역 함수에 의한 연산자 오버로딩 (friend 사용)

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

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

    friend CPoint operator+(const CPoint &Po1, const CPoint &Po2); // friend
};

CPoint operator+(const CPoint &Po1, const CPoint &Po2) // + 연산자 전역 함수
{
    return CPoint(Po1.x + Po2.x, Po1.y + Po2.y);        // private 접근 가능
}

int main(void)
{
    CPoint P1(1, 1);
    CPoint P2(2, 2);
    CPoint P3 = P1 + P2;

    P3.Print();

    return 0;
}
```

멤버 함수에 의한 오버로딩과
전역함수에 의한 연산자 오버로딩이 동시에
존재하는 경우의 호출 우선 순위
멤버 함수 → 전역 함수

3. 연산자 오버로딩 시 고려 사항

1. 기본 타입에 대한 의미 변경 불가
   - intA + intB // 불가능
   - ObjectA + intA // 가능
   - intA + ObjectA // 가능
   - ObjectA + ObjectB // 가능
2. 연산자 오비로딩이 불가능한 연산자
   - . .* :: ?:
   - 연산자 오비로딩이 가능한 연산자

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

3. C++ 연산자 이외의 연산자에 대한 오버로딩 불가
   - P1 @ P2 // 불가능

3. 연산자 오버로딩 시 고려 사항

4. 대입 연산자, 주소 연산자
   - 연산자 오버로딩을 하지 않아도 사용 가능
   - = : 멤버 단위 복사, & : 해당 객체의 주소 반환
5. 연산자 우선 순위 변경 불가
   - P1 + P2 * P3 : * 연산 우선
6. 결합 법칙 변경 불가
   - P1 + P2 + P3 : 결합 법칙은 왼쪽에서 오른쪽으로
7. 피연산자 개수 변경 불가
   - 단항 연산자(++, --) : 하나의 피연산자 요구
   - 곱셈 연산자(*) : 두 개의 피연산자 요구
8. 디폴트 매개변수 불가
   - 디폴트 매개변수가 가능하다면

```
CPoint operator+(const CPoint &Po = CPoint(0, 0)) { ... }
CPoint P3 = P1 +;
```

7. 피연산자 개수 변경 불가 위배

4. 이항 연산자 오버로딩

- 뺄셈(-), 곱셈(*), 나눗셈(/) 연산자 오버로딩
  - 기본 원리는 덧셈(+) 연산자 오버로딩과 동일
- 예 : 다음 연산이 가능하도록 연산자 오버로딩 작성 (a : int형 변수)

① Po1 + Po2 : CPoint(Po1.x + Po2.x, Po1.y + Po2.y) 객체 반환

② Po1 + a : CPoint(Po1.x + a, Po1.y + a) 객체 반환

③ Po1 – Po2 : CPoint(Po1.x – Po2.x, Po1.y – Po2.y) 객체 반환

④ Po1 – a : CPoint(Po1.x – a, Po1.y – a) 객체 반환

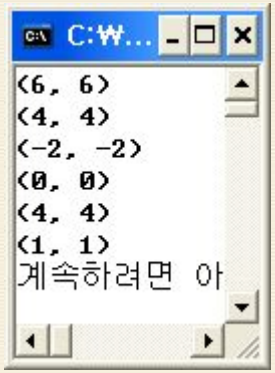
⑤ Po1 * a : CPoint(Po1.x * a, Po1.y * a) 객체 반환

⑥ Po1 / a : CPoint(Po1.x / a, Po1.y / a) 객체 반환

4. 이항 연산자 오버로딩

```
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); }
    CPoint operator+(int a) { return CPoint(x + a, y + a); }
    CPoint operator-(const CPoint &Po) { return CPoint(x - Po.x, y - Po.y); }
    CPoint operator-(int a) { return CPoint(x - a, y - a); }
    CPoint operator*(int a) { return CPoint(x * a, y * a); }
    CPoint operator/(int a) { return CPoint(x / a, y / a); }
    void Print(CPoint Po = CPoint(0, 0))
        { cout << "(" << x << ", " << y << ")"
};
```

```
int main(void)
{
    CPoint P1(2, 2), P2(4, 4);
    int a = 2;

    (P1 + P2).Print();
    (P1 + a).Print();
    (P1 - P2).Print();
    (P1 - a).Print();
    (P1 * a).Print();
    (P1 / a).Print();

    return 0;
}
```



임시 객체를 이용한
멤버 함수 호출

5. 이항 연산자에서 피연산자의 교환 문제

- 앞의 예 : 다음 연산의 교환법칙 성립 여부 및 수행 불가능 여부는?
  - Po1 + Po2 → Po2 + Po1 : 교환법칙 성립
  - Po1 + a → a + Po1 : 교환법칙 성립, 수행 불가능
  - Po1 – Po2 → Po2 – Po1 : 교환법칙 성립X, 수행 가능
  - Po1 – a → a – Po1 : 교환법칙 성립X, 수행 불가능

```
int main(void)
{
    CPoint P1(2, 2), P2(4, 4);
    int a = 2;

    CPoint P3 = a + P1;       // 수행 불가능
    CPoint P4 = a - P2;       // 수행 불가능

    P3.Print();
    P4.Print();

    return 0;
}
```

교환법칙의 성립 여부는
우리의 관심사가 아님.
수행 불가능한 연산을
수행 가능토록 만들려면?

수행 가능하도록
만들어 보자.
a.operator+(P1); // ? 불가능
→ 전역 함수에 의한 연산자 오버로딩

5. 이항 연산자에서 피연산자의 교환 문제

- (a + P1), (a – P1)이 가능하도록 CPoint 클래스 수정

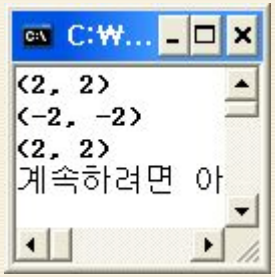
```
class CPoint {
private :
    int x, y;

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); }
    CPoint operator+(int a) { return CPoint(x + a, y + a); }
    CPoint operator-(const CPoint &Po) { return CPoint(x - Po.x, y - Po.y); }
    CPoint operator-(int a) { return CPoint(x - a, y - a); }
    CPoint operator*(int a) { return CPoint(x * a, y * a); }
    CPoint operator/(int a) { return CPoint(x / a, y / a); }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }

    friend CPoint operator+(int a, const CPoint &Po);
    friend CPoint operator-(int a, const CPoint &Po);
};

CPoint operator+(int a, const CPoint &Po)  // a + P1
{
    return CPoint(a + Po.x, a + Po.y);
}

CPoint operator-(int a, const CPoint &Po)  // a - P1
{
    return CPoint(a - Po.x, a - Po.y);
}
```

6. 단항 연산자 오버로딩

단항 연산자 오버로딩

- 기본 원리는 이항 연산자와 동일
  - 음수 부호(-) 연산자 오버로딩
    - ✓ int a = 3;
    - ✓ int b = -a; // a의 값은 변화 없음. b의 값은 -3.
    - ✓ -Po → Po.operator-() → 멤버 함수 operator-(void) 작성

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

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




```
int main(void)
{
    CPoint P1(2, 2);
    CPoint P2 = -P1;      // P1 (2, 2), P2 (-2, -2)
    CPoint P3 = -(-P1);   // P1 (2, 2), P3 (2, 2)

    P1.Print();   P2.Print();   P3.Print();
    return 0;
}
```

6. 단항 연산자 오버로딩

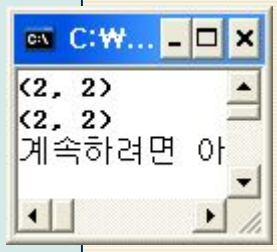
응용 : 음수 부호(–) 연산자의 의미 변경

- –Po → Po의 x, y 값의 부호가 바뀌도록 하려면?
  - CPoint P1(2, 2);
  - CPoint P2 = –(–P1); // P1 (2, 2), P2 (2, 2)가 되어야 함
- 다음과 같은 연산자 오버로딩?
  - CPoint operator–() { x = –x; y = –y; return CPoint(x, y); }
  - CPoint P2 = –P1; // P1 (–2, –2), P2 (–2, –2) → OK!
  - CPoint P2 = –(–P1); // P1 (–2, –2), P2 (2, 2) → P1 No!
    - ✓ 두 번째 –는 임시 객체에 대해 수행되기 때문에 P1에는 영향을 미치지 않음
    - ✓ 해결 방안 : 참조 반환

6. 단항 연산자 오버로딩

- 음수 부호(-) 연산자 오버로딩 (해당 객체의 x, y 부호 변경)

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

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

int main(void)
{
    CPoint P1(2, 2);
    CPoint P2 = -(-P1);       // P1 (2, 2) 두 번 변함, P2 (2, 2)

    P1.Print();
    P2.Print();

    return 0;
}
```



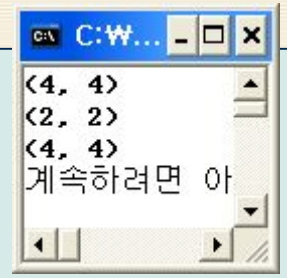
7. 증가, 감소 단항 연산자 오버로딩

- int형 변수에 대한 증가, 감소 연산자 적용
  - 전위 증가 연산자
    - int a = 1;
    - int b = ++a; // a : 2, b : 2 → a가 먼저 증가된 후에 결과를 대입
  - 후위 증가 연산자
    - int a = 1;
    - int b = a++; // a : 1, b : 2 → a가 증가되지만 대입은 그 전 값 사용
- CPoint 클래스에 대한 전위 증가 연산자 오버로딩
  - CPoint P1(1, 1);
  - CPoint P2 = ++P1; // P1 (2, 2), P2 (2, 2)
  - CPoint P3 = ++(++P1); // P1 (4, 4), P3 (4, 4)
  - 이와 같이 되도록 전위 증가 연산자 오버로딩 구현

7. 증가, 감소 단항 연산자 오버로딩

++ 전위 증가 연산자 오버로딩

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

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

int main(void)
{
    CPoint P1(1, 1);
    CPoint P2 = ++P1;     // P1 (2, 2), P2 (2, 2)
    CPoint P3 = ++(++P1); // P1 (4, 4), P3 (4, 4)

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```

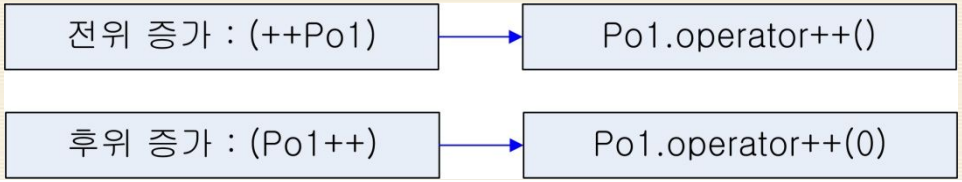


호출한 객체의 참조 전달을 통해 연속적인 ++ 연산자 적용 가능

7. 증가, 감소 단항 연산자 오버로딩

- ++ 후위 증가 연산자
  - CPoint P1(1, 1);
  - CPoint P2 = P1++; // P1 (2, 2), P2 (1, 1)
  - CPoint P3 = (P1++)++; // 원칙적으로는(int형에서) 불가능
    - 증가 연산자를 적용할 수 있는 피연산자로는 l-value만 허용
      - ✓ (++3), (3++), ((a+b)++), (++(a+b)) : 모두 허용하지 않음
      - ✓ ++(++a) : 가능 – 전위 증가 연산의 결과로 l-value인 a 반환
      - ✓ (a++)++ : 불가능 – 후위 증가 연산의 결과로 r-value인 a의 값 반환
    - 다음 중 가능한 것은?
      - ✓ (++a)++;
      - ✓ ++(a++);
    - CPoint 클래스의 경우 전위, 후위 증가 연산자 모두 연속 적용이 가능하도록 만들 수도 있음
  - 후위 증가 연산자 오버로딩
    - Po++ → Po.operator++() → 전위 증가 연산자와 구별 안됨

      → Po.operator++(0)으로 해석 → operator++(int) 멤버 함수 작성!



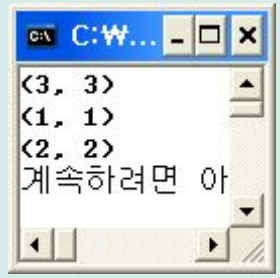
7. 증가, 감소 단항 연산자 오버로딩

++ 후위 증가 연산자 오버로딩

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

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    CPoint operator++(int NotUsed)          // 후위 증가 연산자
        { CPoint temp = (*this); x++; y++; return temp; }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

int main(void)
{
    CPoint P1(1, 1);
    CPoint P2 = P1++;          // P1 (2, 2), P2 (1, 1)
    CPoint P3 = (P1++)++;      // P1 (3, 3), P3 (2, 2)

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```

이전 값을 저장한 후 x, y 값 증가
이전 값 반환

수행은 가능하나 논리적으로
수행 결과가 맞지 않음



8. 입출력 연산자 오버로딩을 이용한 cin, cout의 구현

- print와 scanf를 사용하여 cin, cout 객체 만들기

```
#include <cstdio>
using namespace std;      // VC++ 6.0에서는 삭제

char *endl = "₩n";
char *tab = "₩t";

class ostream {
public :
    ostream &operator<<(int val) {  // int 값에 대한 출력 연산자(<<) 오버로딩
        printf("%d", val);
        return (*this);
    }
    ostream &operator<<(char *str) {// char * 값에 대한 << 연산자 오버로딩
        printf("%s", str);
        return (*this);
    }
};
```

출력(cout)을 위한 클래스

참조 반환 : 연속 << 적용 가능

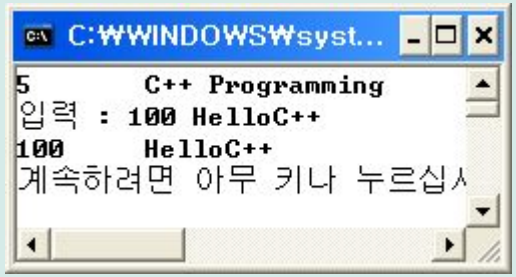
8. 입출력 연산자 오버로딩을 이용한 cin, cout의 구현

- 코드 계속

입력(cin)을 위한 클래스

```
class istream {
public :
    istream &operator>>(int &val) {     // int 값에 대한 >> 연산자 오버로딩
        scanf("%d", &val);
        return (*this);
    }
    istream &operator>>(char *str) {// char * 값에 대한 >> 연산자 오버로딩
        scanf("%s", str);
        return (*this);
    }
};

ostream cout;
istream cin;

int main(void)
{
    int a = 5;
    char str[] = "C++ Programming";

    cout << a << tab << str << endl;

    cout << "입력 : ";
    cin >> a >> str;

    cout << a << tab << str << endl;

    return 0;
}
```

입출력 객체 생성



연속 출력 가능

istream과 ostream 구현 원리 이해
C++에 포함된 cin, cout에 대한 구체적인 사용 방법은 11장에서 설명

9. friend 함수를 사용한 입출력 연산자 오버로딩

- CPoint 클래스 객체에 대한 출력 연산자 사용
  - CPoint P1(3, 4); cout << P1; // 가능하게 하려면?

```
#include <cstdio>
using namespace std;      // VC++ 6.0에서는 삭제

class CPoint {
private :
    int x, y;

public :
    CPoint(int a, int b) : x(a), y(b) { }

    friend class ostream;
};

class ostream {
public :
    ostream &operator<<(const CPoint &Po) { // CPoint 객체의 << 오버로딩
        printf("(%d, %d)₩n", Po.x, Po.y);
        return (*this);
    }
};
```

문제점 : C++ 라이브러리의 **ostream** 클래스를 수정하기 어려움!
→ 라이브러리의 **ostream** 클래스를 수정하지 않고 가능하게!
→ 전역 함수에 의한 연산자 오버로딩

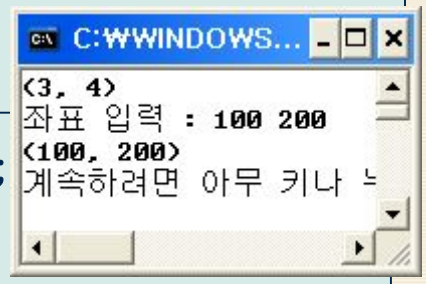
9. friend 함수를 사용한 입출력 연산자 오버로딩

- 전역 함수에 의한 <<, >> 연산자 오버로딩
  - cout << P1; → operator<<(cout, P1);

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

class CPoint {
private :
    int x, y;

public :
    CPoint(int a, int b) : x(a), y(b) { }

    friend ostream &operator<<(ostream &out, const CPoint Po);
    friend istream &operator>>(istream &in, CPoint &Po);
};

ostream &operator<<(ostream &out, const CPoint Po)// 전역 함수 << 오버로딩
{
    out << "(" << Po.x << ", " << Po.y << ")" << endl;
    return out;
}

istream &operator>>(istream &in, CPoint &Po)      // 전역 함수 >> 오버로딩
{
    cout << "좌표 입력 : ";
    in >> Po.x >> Po.y;
    return in;
}
```

cout의 참조 반환 : 연속적인 << 가능

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

    cin >> P1;
    cout << P1;

}
```



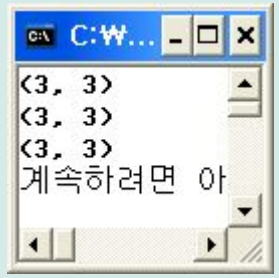
10. 대입 연산자 오버로딩

- 디폴트 대입 연산자의 모양
  - 안 1 : void operator=(const CPoint &Po) { x = Po.x; y = Po.y; }
    - P1 = P2; // 수행 가능
    - P1 = P2 = P3; // 수행 불가능
  - 안 2 : CPoint operator=(const CPoint &Po) { x = Po.x; y = Po.y; return Po; }
  - 안 3 : CPoint &operator=(const CPoint &Po) { x = Po.x; y = Po.y; return (*this); }
  - 안 4 : const CPoint &operator=(const CPoint &Po) { x = Po.x; y = Po.y; return Po; }
  - 안 3이 가장 효율적으로 수행될 수 있음
    - 실제 디폴트 대입 연산자의 모양은 안 3과 동일
    - (P1 = P2) = P3; 실행 가능 (int 개념과 동일)
  - 주의 사항 : 대입 연산자는 멤버 함수에 의한 오버로딩만 가능
    - 멤버 함수에 의한 오버로딩만 가능한 연산자 : =, [], ->

10. 대입 연산자 오버로딩

⌗ 디폴트 대입 연산자의 명시적 구현

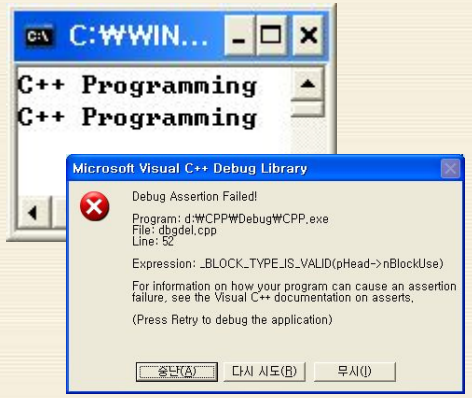
```
class CPoint {
private :
    int x, y;

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

int main(void)
{
    CPoint P1(1, 1), P2(2, 2), P3(3, 3);
    P1 = P2;                // 대입 연산
    P1 = P2 = P3;           // 연속적인 대입 연산

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```



10. 대입 연산자 오버로딩

- CString 클래스 객체의 대입 시 문제점은?

```
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; }
};
```

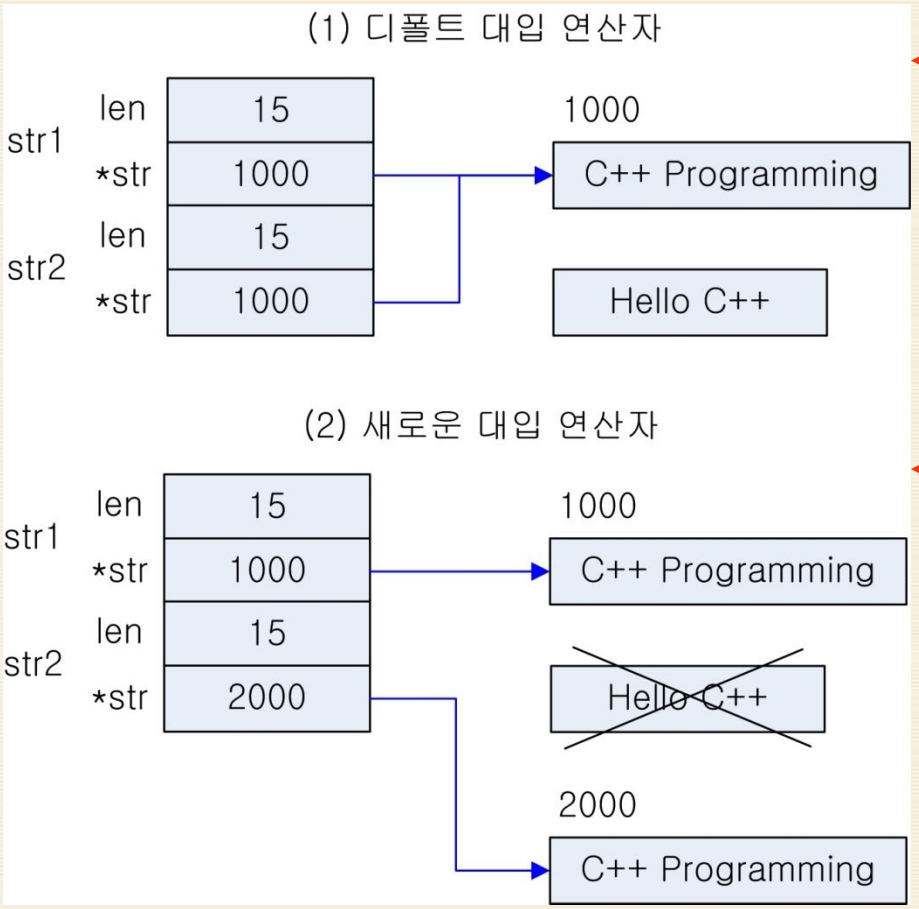



```
int main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = "Hello C++";

    str2 = str1;   // 대입 연산

    str1.Print();
    str2.Print();

    return 0;
}
```

10. 대입 연산자 오버로딩

⌗ CString 객체 대입의 동작

str2 = str1;의 수행 결과



멤버 단위 복사

동일한 메모리를 가리킴
→ 소멸자 수행 시 에러 발생

원하는 동작

별도의 메모리를 가리키도록
대입 연산자 오버로딩 구현!

10. 대입 연산자 오버로딩

- CString 클래스에 대한 명시적 대입 연산자 오버로딩

```
#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 &operator=(const CString &string) {    // 대입 연산자 오버로딩
        delete [] str;
        len = string.len;
        str = new char[len + 1];
        strcpy(str, string.str);
        return (*this);
    }
    ~CString() { delete [] str; }
    void Print() { cout << str << endl; }
};
```

연속적인 대입이 가능하도록 참조 반환

10. 대입 연산자 오버로딩

코드 계속

```
int main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = "Hello C++";
    CString str3 = "Operator Overloading";

    str1 = str2 = str3;      // 연속적인 대입 연산

    str1.Print();
    str2.Print();
    str3.Print();

    return 0;
}
```

11. 배열 첨자 연산자 오버로딩

- int형 배열의 예
  - int ary[10];
  - int a = ary[0]; // 배열 첨자 연산자 : 첫 번째 원소의 값
  - ary[3] = 5; // 배열 첨자 연산자 : 네 번째 원소(int형 변수 그 자체)
- CPoint 클래스의 예
  - int a = Po[0]; // Po의 x 값(으로 가정)
  - Po[1] = 5; // Po의 y 값(으로 가정)을 5로 변경
  - 배열 첨자 연산자 오버로딩
    - Po[0] → Po.operator[](0)

```
int operator[](int index) {  // index는 0 또는 1이라고 가정
    if (index == 0) return x;
    else if (index == 1) return y;
}
```

문제점
Po[1] = 5; 가 수행되지 않음
해결
참조 반환
int &operator …

11. 배열 첨자 연산자 오버로딩

- CPoint 클래스의 배열 첨자 연산자 오버로딩

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

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    int &operator[](int index) {    // 배열 첨자 연산자 오버로딩, 참조 반환
        if (index == 0) return x;
        else if (index == 1) return y;
    }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

int main(void)
{
    CPoint P1(1, 1);
    P1[0] = 2;     // P1의 x 변수 그 자체
    P1[1] = 3;     // P1의 y 변수 그 자체

    P1.Print();

    return 0;
}
```