인터넷미디어전공 실습 II
(C++)

강의노트 #9: 연산자 오버로딩

2006. 5. 24.

담당교수: 조 재 수

E-mail: jaesoo27@kut.ac.kr

학습 내용

- 연산자 오버로딩의 기초
- 연산자 오버로딩의 의미
- 연산자 오버로딩의 두가지 방법
  - 멤버함수를 이용한 방법
  - 전역함수를 이용한 방법(friend 함수)
- 이항 연산자 오버로딩
- 관계 연산자와 논리 연산자 오버로딩
- 단항 연산자 오버로딩
- 대입 연산자 오버로딩
- 교환법칙 해결하기
- cout, cin, endl 이해하기
- <<, >> 오버로딩

1. 연산자 오버로딩의 기초

- 연산자 오버로딩(operator overloading)은 함수 오버로딩 (function overloading)의 한 유형이다.
- 연산자가 오버로딩 될 때는 연산자의 고유한 의미가 없어진다.
- 연산자를 오버로딩 하려면 연산자 함수(operator function)를 생성해야 한다.
- 연산자 함수는 그것이 정의된 클래스의 멤버함수로 선언되거나 friend로 선언된 전역함수이다.
- 멤버 연산자 함수와 friend 연산자 함수 사이에는 약간의 차이가 있다.
- 멤버 연산자 함수의 일반적인 형식은 다음과 같다.

```
return-type class-name::operator #(arg-list)
{
        // 수행되어야 할 연산
}
```


1. 연산자 오버로딩의 기초

- 대부분의 경우에서는 연산자 함수의 반환 형(return-type)은 그것이 정의된 클래스가 된다.
- 오버로딩된 연산자는 # 위치에 놓인다.
- arg-list의 내용은 연산자 함수를 구현하는 방법과 오버로딩된 연산자의 형태에 따라 바뀐다.

- 연산자를 오버로딩할 때의 두 가지 중요한 제약

  1) 연산자의 우선 순위를 변경할 수 없다.

  2) 연산자의 피연산자 수를 변경할 수 없다.

- 대부분의 C++ 연산자는 오버로딩 될 수 있다. 그러나 다음 연산자를 오버로딩할 수 없다.

  . :: .* ?

  또한 선행 처리 연산자(preprocessor operator)도 오버로딩할 수 없다.

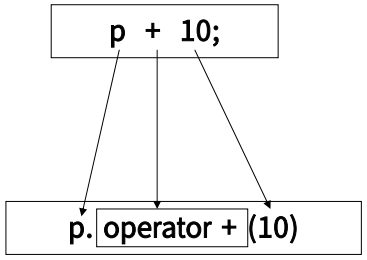
2. 연산자 오버로딩의 의미

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

class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    void operator+(int val);
};

void Point::ShowPosition() {
    cout<<x<<" "<<y<<endl;
}

void Point::operator+(int val) {
    x+=val;
    y+=val;
}
```

```
int main(void)
{
    Point p(3, 4);
    p.ShowPosition();

    p.operator+(10);
   // p+10;
    p.ShowPosition();
    return 0;
}
```

- p.operator+(10) 와 p+10은 같은 표현이다.
- operator라는 키워드와 연산자 기호를 묶어서 함수의 이름을 정의하면, 함수의 이름을 이용한 함수 호출뿐만 아니라, 연산자를 이용한 함수 호출도 허용

2. 연산자 오버로딩의 의미



- p+10 이라는 문장과 p.operator+(10) 이라는 문장은 100% 같은 문장이다.
- p+10 이라는 문장은 p.operator+(10) 이라는 문장으로 해석된다. 이러한 것을 멤버함수에 의한 연산자 오버로딩에 해당된다.

3. 연산자 오버로딩 두 가지 방법

- 멤버 함수에 의한 연산자 오버로딩

```
class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    Point operator+(const Point& p);
};

void Point::ShowPosition(){
    cout<<x<<" "<<y<<endl;
}

Point Point::operator+(const Point& p){
    Point temp(x+p.x, y+p.y);
    return temp;
}
```

```
int main(void)
{
    Point p1(1, 2);
    Point p2(2, 1);
    Point p3=p1+p2;
    p3.ShowPosition();

    return 0;
}
```


3. 연산자 오버로딩 두 가지 방법

- 전역 함수에 의한 연산자 오버로딩

```
class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    friend Point operator+(const Point&, const Point&);
};
void Point::ShowPosition(){
    cout<<x<<" "<<y<<endl;
}

Point operator+(const Point& p1, const Point& p2)
{
   Point temp(p1.x+p2.x, p1.y+p2.y);

   return temp;
}
```

```
int main(void)
{
    Point p1(1, 2);
    Point p2(2, 1);
    Point p3=p1+p2;
    p3.ShowPosition();

    return 0;
}
```

3. 연산자 오버로딩 두 가지 방법

- 연산자 오버로딩의 주의 사항
  - 본 의도를 벗어난 연산자 오버로딩!
  - 연산자 우선 순위와 결합성은 유지된다.
  - 디폴트 매개변수 설정이 불가능하다.
  - 디폴트 연산자들의 기본 기능 변경 불가

```
int operator+(int a, int b)      // 정의 불가능한 함수
{
   return a+b+3;
}
```


4. 이항연산자의 오버로딩

- 멤버 연산자 함수가 이항 연산자(binary operator)를 오버로딩할 때 단지 한 개의 매개변수만을 갖는다.
- 이 매개변수는 연산자의 오른쪽에 있는 객체를 받게 된다.
- 왼쪽에 있는 객체는 연산자 함수를 호출하는 객체이며 this 포인터를 묵시적으로 전달 받는다.

```
// Ex01.cpp
// coord 클래스에 대해 + 연산자를 오버로딩한다.

#include <iostream>
 using namespace std;

 class coord {
                int x, y; // 좌표 값
   public:
      coord() { x=0; y=0; }
      coord(int i, int j) { x=i; y=j ; }
      void get_xy(int &i, int &j) { i=x; j=y; }
      coord operator+(coord ob2);
};

// coord 클래스에 대해 +를 오버로딩한다.
coord coord::operator+(coord ob2)
{
       coord temp;
       temp.x = x + ob2.x;
       temp.y = y + ob2.y;

       return temp;
}
```

```
int main()
{
    coord o1(10, 10), o2(5, 3), o3;
     int x, y;

     o3 = o1 + o2;
   // 두 객체를 더한다.
 // operator+()를 호출한다.

    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ",
    Y: " << y << " \ n";

     return 0;
}

// 출력 결과
  (o1+o2) X: 15, Y: 13
```


4. 이항 연산자의 오버로딩

- operator+( ) 함수는 x에 피연산자의 X 좌표의 합을 저장하고 y에 Y 좌표의 합을 저장한 coord 형의 객체를 반환한다.
- temp라는 임시 객체가 operator+( ) 내에서 결과를 저장하기 위해 사용되고, 이 객체를 반환하는 것에 주의하라.
- 이 예에서 coord 객체가 반환되기 때문에 다음 문장 역시 타당하다.

```
(o1+o2).get_xy(x, y);
```

```
// coord 클래스에 대해 +, -, = 연산자를 오버로딩 한다.
// Ex01-1.cpp
#include <iostream>
using namespace std;
class coord {
        int x, y; // 좌표 값
public:
        coord() { x=0; y=0; }
        coord(int i, int j) { x=i; y=j; }
        void get_xy(int &i, int &j) { i=x; j=y; }
        coord operator+(coord ob2);
        coord operator-(coord ob2);
        coord operator=(coord ob2);
};
// coord 클래스에 대해 +를 오버로딩한다.
coord coord::operator+(coord ob2)
{
        coord temp;
        temp.x = x + ob2.x;
        temp.y = y + ob2.y;
        return temp;
}
// coord 클래스에 대해 -를 오버로딩한다.
coord coord::operator-(coord ob2)
{
        coord temp;
        temp.x = x - ob2.x;
        temp.y = y - ob2.y;
        return temp;
}
```

```
// coord 클래스에 대해 =을 오버로딩한다.
coord coord::operator=(coord ob2)
{
    x = ob2.x;
    y = ob2.y;
    return *this;  // 대입된 객체를 반환한다.
}

int main ()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;
    o3 = o1+ o2; // 두 객체를 더한다. operator+()를 호출한다.
    o3.get_xy(x, y);
    cout << "(o1+o2) x: " << x << ", y: " << y << "\n";
    o3 = o1 - o2; // 두 객체를 뺀다.
    o3.get_xy(x, y);
    cout << "(o1-o2) x: " << x << ", y: " << y << "\n";
    o3 = o1; // 하나의 객체를 대입한다.
    o3.get_xy(x, y);
    cout << "(o3=o1) x: " << x << ", y: " << y << "\n";
    return 0;
}
```


4. 이항 연산자의 오버로딩(계속)

- operator-( ) 함수는 operator+( ) 함수와 비슷하게 구현되어 있다.
- 위의 예는 피연산자의 순서가 중요한 연산자를 오버로딩할 때 주의해야 할 점을 보여준다.
- operator+( ) 함수를 만들 때, 피연산자의 순서는 중요하지 않다(즉, A+B나 B+A는 같다).
- 뺄셈 연산에서는 순서에 따라 다른 결과가 나온다.
- 뺄셈 연산자를 정확히 오버로딩하기 위해서 왼쪽 피연산자에서 오른쪽 피연산자를 빼는 것이 필요하다. 왜냐하면 왼쪽 피연산자가 operator-( ) 를 호출하기 때문에, 뺄셈은 다음 순서가 되어야 한다.

  x - ob2.x;

- 이항 연산자가 오버로딩될 때, 좌측 피연산자는 묵시적으로 함수에 전달되고 우측 피연산자는 인수로서 전달된다.

4. 이항 연산자의 오버로딩(계속)

- 대입 연산자 함수에서, 첫 번째 주의사항은 왼쪽 피연산자(즉, 값이 대입되는 객체)는 연산에 의해 변경된다는 것이다. 이것은 통상적인 대입의 의미와 같다.
- 두 번째 주의사항은 연산자 함수가 *this를 반환한다는 것이다. 즉, operator=( ) 함수는 값이 대입되는 객체를 반환한다. 이것은 연속적인 대입이 가능하도록 하기 위해서이다. C++에서 다음 문장은 문법적으로 옳은 것이며 매우 자주 사용된다.

  a = b = c = d = 0;

- *this를 반환함으로서 오버로딩된 대입 연산자는 coord 형의 객체를 위와 비슷한 방법으로 사용할 수 있다. 예를 들면, 다음 문장은 타당하다.

  o3 = o2 = o1;

- 오버로딩된 대입 연산자 함수가 대입된 객체를 반환해야 한다는 규칙은 없지만, 내장형에 대해서 작동하는 방식대로 오버로딩된 =이 해당 클래스에 대해서 작동하기를 원한다면 *this를 반환해야 한다.


```
// Ex01-2.cpp
// ob + int 와 ob + ob를 위해 +를 오버로딩 한다.

    #include <iostream>
    using namespace std;

    class coord {
                int x, y; // 좌표 값
      public:
          coord() { x = 0; y = 0; }
          coord(int i, int j) { x = i; y = j; }
          void get_xy(int &i, int &j) { i = x; j = y; }
          coord operator+(coord ob2); // ob + ob
          coord operator+(int i); // ob + int
      };

    // coord 클래스에 대해 +를 오버로딩한다.
    coord coord::operator+(coord ob2)
    {
          coord temp;

          temp.x = x + ob2.x;
          temp.y = y + ob2.y;

          return temp;
    }
```

```
// ob + int를 위해 +를 오버로딩한다.
  coord coord::operator+(int i)
  {
          coord temp;
          temp.x = x + i;
          temp.y = y + i;
          return temp;
   }
 int main()
  {
       coord o1(10, 10), o2(5, 3), o3;
       int x, y;
 // 두 객체를 더한다. operator+(coord)를 호출한다.
       o3 = o1 + o2;
       o3.get_xy(x, y);
       cout << "(o1+100) x: " << x << ", y: " << y
    << " \ n";
  // 객체에 int형 값을 더한다.   operator+(int)를 호출한다.
       o3 = o1 + 100;
       o3.get_xy(x, y);
       cout << "(o1+100) X: " << x << ", Y: " << y
    << " \ n";
        return 0;
  }
```

4. 이항 연산자의 오버로딩 (계속)

- 내장형을 포함한 객체를 연산에서 사용하기 위해서는, 멤버 연산자 함수를 오버로딩할 때 내장형은 연산자의 오른쪽에 있어야 한다.
- 그 이유는 연산자 함수를 호출하는 객체는 왼쪽에 있어야 하기 때문이다. 그 예로서, 다음 문장에 대해 컴파일러는 어떻게 반응하는지를 살펴보자.

```
o3 = 19 + o1;      // int 형 값 + 객체
```

- 오버로딩된 operator+(int i) 함수는 객체가 왼쪽에 있을 경우에만 작동한다. 따라서, 이 문장은 컴파일 에러를 유발한다.

(Error message: binary '+' : no global operator defined which takes type 'class coord' (or there is no acceptable conversion))

4. 이항 연산자의 오버로딩(계속)

- 연산자 함수에서 참조 매개변수를 사용할 수 있다. 다음 예는 coord 클래스에 대해 + 연산자를 오버로딩하는 방법이다.

```
// 참조를 사용하여 coord 클래스에 대해 +를 오버로딩한다.
coord coord::operator+(coord &ob2)
{
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}
```

- 연산자 함수에서 참조 매개변수를 사용하는 이유는 효율성 때문이다. 객체를 매개변수로 함수에 전달하는 방법은 많은 오버헤드(overhead)를 발생시키고 CPU를 상당히 잡아 먹는다.
- 객체의 주소를 전달하는 것은 언제나 빠르고 효율적이다. 연산자를 자주 사용한다면, 참조 매개변수를 사용하는 것이 일반적으로 성능을 증진시킨다.

4. 이항 연산자의 오버로딩(계속)

- 참조 매개변수를 사용하는 또 다른 이유는 피연산자의 복사본이 소멸할 때 일어나는 문제점을 피하기 위해서이다.
- 인수가 값에 의해 전달(call by value) 될 때, 그 인수의 복사본이 만들어진다.
- 객체가 소멸자를 가진다면, 함수가 종료될 때 복사본의 소멸자가 호출된다. 어떤 경우에는 소멸자는 호출한 객체가 필요로 하는 것을 소멸시킬 수도 있다. 이런 경우에는 값에 의한 매개변수 전달 대신에 참조 매개 변수를 사용하는 것이 문제를 간단하면서도 효율적으로 해결하는 방법이다. 물론, 복사 생성자를 정의해서 이 문제를 방지할 수도 있다.

연습문제 9-1

1. Point 객체를 이용한 덧셈 연산이 가능하도록 연산자 오버로딩 되어 있었다. 이번에는 뺄셈 연산도 가능하도록 연산자를 멤버함수에 의한 방식으로 오버로딩 하여라.

main 함수의 예

```
int main(void){
   point p1(4, 2);
   point p2(2, 1);
   point p3 = p1-p2; // p1.operator-(p2);
   p3.ShowPosition();
   return 0;
}
```

연습문제 9-2

2. Point 객체를 기반으로, 두 객체의 데이터를 비교할 수 있는 != 연산자를 멤버 함수에 의한 방식으로 오버로딩하여라.

Main 함수의 예

```
int main(void)
{
    Point p1(2, 1);
    Point p2(2, 1);
    Point p3(3, 3);
   if(p1 != p2) cout << "다르다!" << endl;
   else  cout << "같다!" << endl;
   if(p2 != p3) cout << "다르다!" << endl;
   else cout << "같다!" << endl;
   return 0;
}
```


연습문제 9-3

3. coord에 대해 *와 / 연산자를 오버로딩하고, 이것이 작동하는 예제 프로그램을 작성하라.

5. 관계 연산자와 논리 연산자의 오버로딩

- 관계 연산자(relational operator)와 논리 연산자(logical operator)도 오버로딩할 수 있다.
- 관계 연산자와 논리 연산자가 원래 의미대로 동작하도록 이 연산자들을 오버로딩 한다면, 연산자 함수가 정의된 클래스의 객체가 아니라 참 또는 거짓을 나타내는 정수를 반환하기를 바랄 것이다. 이렇게 함으로써 연산자가 참 또는 거짓을 반환할 수 있을 뿐만 아니라 다른 자료형을 포함하는 관계식과 논리식에서 사용될 수 있다.
- (주의) 최신의 C++ 컴파일러를 사용하고 있다면, bool 형의 값을 반환하는 오버로딩 된 관계 연산자와 논리 연산자 함수를 사용할 수 있지만, 그렇게 하는 것은 별 이득이 없다. bool 형은 true와 false의 두 가지 값만을 정의한다. 이 값들은 자동으로 0이 아닌 값과 0의 값으로 변환된다. 또한 0이 아닌 값과 0의 값은 자동으로 true와 false로 변환된다.


```
// Ex02.cpp
// 다음 프로그램서 = =와 && 연산자가 오버로딩되어 있다.
// coord 클래스에 관련된 ==와 &&을 오버로딩한다.

#include <iostream>
using namespace std;
class coord {
        int x, y; // 좌표 값
  public:
        coord() { x=0; y=0; }
        coord(int i, int j) { x=i; y=j; }
        void get_xy(int &i, int &j) { i=x; j=y; }
        int operator==(coord ob2);
        int operator&&(coord ob2);
};
// coord를 위해 == 연산자를 오버로딩한다.
int coord::operator==(coord ob2){
  return x==ob2.x && y==ob2.y;  }
// coord를 위해 && 연산자를 오버로딩한다.
int coord::operator&&(coord ob2){
return (x && ob2.x) && (y && ob2.y);}
```

```
int main()
{
coord o1(10, 10), o2(5, 3), o3(10, 10),
   o4(0, 0);

 if(o1==o2) cout << "o1 same as o2 \ n";
else cout << "o1 and o2 differ \ n";

 if(o1==o3) cout << "o1 same as o3 \ n";
 else cout << "o1 and o3 differ \ n";

 if(o1&&o2) cout << "o1 && o2 is
  true \ n";
 else cout << "o1 && o2 is false \ n";

 if(o1&&o4) cout << "o1 && o4 is
  true \ n";
  else cout << "o1 && o4 is false \ n";

  return 0;
}
```

6. 단항 연산자 오버로딩

- 단항 연산자(unary operator)를 오버로딩하는 것은 단 하나의 피연산자만이 있다는 점을 제외하고는 이항 연산자(binary operator)를 오버로딩하는 것과 비슷하다.
- 멤버 함수를 사용하여 단항 연산자를 오버로딩할 때, 함수는 매개변수를 갖지 않는다. 즉, 피연산자가 하나이기 때문에 바로 이 피연산자가 연산자 함수를 호출한다.
- 다른 매개변수는 필요 없다.


```
//Ex04.cpp
//  다음 프로그램은 coord 클래스에 대해 증
    가 연산자(++)를 오버로딩한다..
// coord 클래스에 관련된 ++를 오버로딩한다.

 #include <iostream>
 using namespace std;
 class coord {
                int x, y;    // 좌표 값
   public:
     coord() { x=0; y=0; }
     coord(int i, int j) { x=i; y=j; }
     void get_xy(int &i, int &j) { i=x; j=y; }
     coord operator++();
 };

 // coord 클래스에 대해 ++를 오버로딩한다.
  coord coord::operator++()
  {
         x++;
         y++;
        return *this;
   }
```

```
int main()
{
     coord o1(10, 10);
     int x, y;

     ++o1; // 객체에 증가 연산을 적용한다.
     o1.get_xy(x, y);
     cout << "(++o1) X: " << x << ", Y: "
   << y << " \n";

      return 0;
  }
```

- 증가 연산자는 피연산자를 1만큼 증가시키기 때문에, 오버로딩 연산자 + +는 연산의 대상이 되는 객체를 변경한다.
- 이 연산자 함수는 증가된 객체를 반환한다. 이것은 증가 연산자가 다음과 같은 문장의 일부로 사용될 수 있게 한다.

  o2 = ++o1;
- 이항 연산자와 마찬가지로, 단항 연산자가 원래의 의미대로 단항 연산자를 오버로딩해야 한다는 규칙은 없지만 대부분의 경우에서는 원래의 의미대로 오버로딩된다.

6. 단항 연산자 오버로딩

- C++의 초기 버전에서 증가 연산자나 감소 연산자를 오버로딩할 때, 오버로딩된 + +나 - -가 피연산자의 앞에 있는지 또는 뒤에 있는지를 결정하는 방법이 없었다. 즉, 위의 프로그램에서 다음 두 문장은 동일한 문장이었다.

```
o1++;
++o1;
```

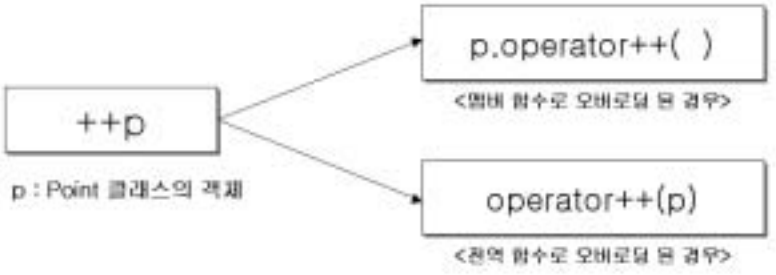
- 최신 C++에서는 컴파일러가 이 두 가지 문장을 구분할 수 있는 방법이 정의되어 있다. 이렇게 하기 위해서는 두 가지 다른 버전의 operator++( ) 함수를 만들어야 한다. 첫 번째 버전은 앞의 예에서와 같이 정의되고, 두 번째 버전은 다음과 같이 선언된다.

```
coord coord::operator++(int notused);
```

- 만약 ++가 피연산자의 앞에 온다면, operator++( ) 함수가 호출된다. 그러나 ++가 피연산자의 뒤에 온다면, operator++(int notused) 함수가 사용된다. 이 경우에 notused에 언제나 0이 전달된다. 그러므로 - -나 ++이 앞에 붙느냐 뒤에 붙느냐에 따른 차이(선행 표기와 후행 표기에 따른 차이)가 클래스 객체에서 중요하다면, 두 연산자 함수를 각각 구현해야 한다.

6. 단항 연산자의 오버로딩

- 증가, 감소 연산자 오버로딩



그림

6. 단항 연산자의 오버로딩

```
class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    Point& operator++();
    friend Point& operator--(Point& p);
};

void Point::ShowPosition(){
    cout<<x<<" "<<y<<endl;
}

Point& Point::operator++(){
    x++;
    y++;
    return *this;
}

Point& operator--(Point& p){
    p.x--;
    p.y--;
    return p;
}
```

```
int main(void)
{
    Point p(1, 2);
    ++p;                    //p의 x, y 값을 1씩 증가
   //p++;                  ++p와 p++의 차이는?
    p.ShowPosition();       //2, 3

    --p;                    //p의 x, y 값을 2씩 증가
    p.ShowPosition();       //1, 2

    ++(++p);
    p.ShowPosition();       //3, 4

    --(--p);
    p.ShowPosition();       //1, 2
    return 0;
}
```


6. 단항 연산자의 오버로딩

- 선 연산과 후 연산의 구분

```
++p  →  p.operator++();   // 선 증가, 후 연산
p++  →  p.operator++(int); // 선 연산, 후 증가
```

```
--p  →  p.operator--();
p--  →  p.operator--(int);
```

6. 단항 연산자의 오버로딩

```
class Point {
private:
    int x, y;
public:
    Point(int _x=0, int _y=0):x(_x), y(_y){}
    void ShowPosition();
    Point& operator++();
    Point operator++(int);
};
void Point::ShowPosition(){
    cout<<x<<" "<<y<<endl;
}

Point& Point::operator++(){
    x++;
    y++;
    return *this;
}
Point Point::operator++(int){
    Point temp(x, y);  // Point temp(*this);
    x++;
    y++;
    return temp;
}
```

```
int main(void)
{
    Point p1(1, 2);
    (p1++).ShowPosition();
    p1.ShowPosition();

    Point p2(1, 2);
    (++p2).ShowPosition();
    return 0;
}
```

출력결과는?

1 2 or 1 2
1 2 2 3
2 3 2 3

6. 단항 연산자 오버로딩

- C++에서 뺄셈 기호(-)는 이항 연산자와 단항 연산자로 사용할 수 있다.
- 클래스에 대해 두 가지 모두로 사용하려면 어떻게 오버로딩 해야 할 것인가?
- 그 해결책은 실제로 아주 간단하다. 단지 두 번 오버로딩 하면 된다. 즉, 한 번은 이항 연산자로서, 또 한 번은 단항 연산자로서 오버로딩 한다.
- 다음 프로그램은 이러한 방법을 보여준다.

```
// coord 클래스에 대해 -를 오버로딩한다.
#include <iostream>
 using namespace std;

class coord {
            int x, y;   // 좌표값
 public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
   coord operator-(coord ob2); // 이항 마이너스
    coord operator-();   // 단항 마이너스
};

// coord 클래스에 대해 이항 -를 오버로딩한다.
coord coord::operator-(coord ob2)
{
      coord temp;

      temp.x = x - ob2.x;
      temp.y = y - ob2.y;

      return temp;
}
```

```
// coord 클래스에 관련된 단항 -를 오버로딩한다.
coord coord::operator-()
{
      x = -x;
      y = -y;
      return *this;
}
 int main()
 {
      coord o1(10, 10), o2(5,7);
       int x, y;

      o1 = o1 - o2; // 뺄셈
       o1.get_xy(x, y);
     cout << "(o1-o2) X: " << x << ", Y: " << y
   << " \ n";

      o1 = -o1; // 부호를 바꾼다.
       o1.get_xy(x, y);
      cout << "(-o1) X: " << x << ", Y: " << Y
   << " \ n";

        return 0;
   }
```


6. 단항 연산자 오버로딩

- ㅡ가 이항 연산자로 오버로딩될 경우에는 하나의 매개변수만을 갖는다.
- ㅡ가 단항 연산자로서 오버로딩될 경우에는 매개변수를 갖지 않는다.
- 매개변수 갯수의 차이 때문에 ㅡ가 두 개의 연산으로 오버로딩 될 수 있다.
- 뺄셈 기호(ㅡ)가 이항 연산자로서 사용된 경우에는 operator-(coord ob2) 함수가 호출되고, 단항 연산자로서 사용된 경우에는 operator-( ) 함수가 호출된다.

연습문제

1. coord 클래스에서 ㅡ를 오버로딩하라. 선행(prefix) 표기와 후행(postfix) 표기 각각에 대해 작성하라.

2. coord 클래스에 대한 + 연산자가 이항 연산자와 단항 연산자 모두로 사용되도록 오버로딩하라. +가 단항 연산자로서 사용될 경우, 음수는 양수가 되도록 하라.

연습문제

3. friend 함수를 사용(전역함수로)하여 coord 클래스에 관련된 ㅡㅡ연산자와 / 연산자를 오버로딩하라.

4. coord 클래스 객체의 각 좌표에 정수를 곱하는 연산을 오버로딩하라.
   ob * nt 혹은 int * ob 중 어느 것을 사용해도 연산이 가능하도록 하라.

7. 대입 연산자

- 클래스에 관련된 대입 연산자를 오버로딩하는 것도 가능하다.
- 디폴트로, 대입 연산자를 객체에 적용할 때는, 우측에 있는 객체가 좌측에 있는 객체에 비트 단위로 복사된다.
- 이것이 사용자가 원하는 것이라면, 사용자가 operator=( ) 함수를 작성할 필요는 없다.

7. 대입 연산자 오버로딩

- 디폴트 대입 연산자
  - 멤버 대 멤버 복사
  - DefaultSubOp.cpp

p1 = p2의 의미는 p1.operator=(p2);

```
Point& Point::operator=(const Point& p)
{
    x=p.x;
    y=p.y;
    return *this;
}
```

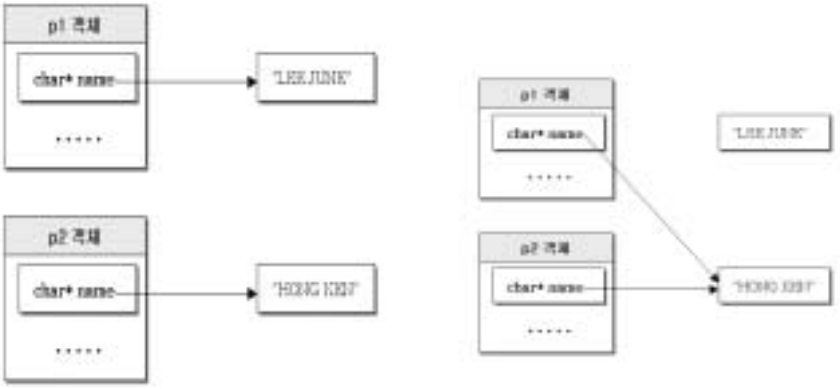
7. 대입 연산자 오버로딩

```
#include <iostream>
using std::endl;
using std::cout;
using std::ostream;

class Person {
private:
	char* name;
public:
	Person(char* _name);
	Person(const Person& p);
	~Person();
	friend ostream& operator<<(ostream& os, const Person&
p);
};

Person::Person(char* _name){
	name= new char[strlen(_name)+1];
	strcpy(name, _name);
}
Person::Person(const Person& p){
	name= new char[strlen(p.name)+1];
	strcpy(name, p.name);
}
Person::~Person(){
	delete[] name;
}
```

```
ostream& operator<<(ostream& os,
const Person& p)
{
os<<p.name;

return os;
}

int main()
{

Person p1("LEE JUNE");

Person p2("HONG
KEN");


cout<<p1<<endl;

cout<<p2<<endl;

    p1=p2;  // 문제의 원인.


cout<<p1<<endl;


return 0;
}
```


7. 대입 연산자 오버로딩

- 디폴트 대입 연산자의 문제점



7. 대입 연산자 오버로딩

- 깊은 복사(Deep Copy)를 하는 대입 연산자

```
Person& Person::operator=(const Person& p)
{
    delete []name;
    name= new char[strlen(p.name)+1];
    strcpy(name, p.name);
    return *this;
}
```



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

class strtype {
                char *p;
                int len;
  public:
        strtype(char *s);
        ~strtype() {
         cout << "Freeing " << (unsigned) p << '\n';
          delete [] p;
        }
        char *get() { return p; }
        strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
{
        int l;

        l = strlen(s)+1
        p = new char [l];
        if (!p) {
              cout << "Allocation error\n";
               exit(1);
        }
        len = l;
        strcpy(p, s);
}
```

```
    // 객체를 대입한다.
strtype &strtype::operator=(strtype &ob)
{
    // 더 많은 메모리가 필요한지 확인한다.
   if(len < ob.len) { // 더 많은 메모리를 할당할 필요가 있다면...
            delete [ ] p;
             p = new char [ob.len];
            if(!p) {
                 cout << "Allocation error\n";
                  exit(1);
            }
       }
       len = ob.len;
       strcpy(p, ob.p);
        return *this;
}

int main()
{
    strtype a("Hello"), b("There");

     cout << a.get() << '\n';
      cout << b.get() << '\n';

       a = b; // 이제 p를 덮어쓰지 않는다.

     cout << a.get() << '\n';
     cout << b.get() << '\n';

     return 0;
   }
```

- 오버로딩된 대입 연산자는 p를 덮어쓰는 것을 방지한다. 좌측에 있는 객체가 대입될 문자열을 저장할만한 충분한 메모리를 할당하였는지를 먼저 검사한다. 만약 할당받지 않았다면 그 메모리는 해제되고 다른 메모리가 할당된다. 그 다음, 문자열은 메모리에 복사되고 그 길이는 len에 복사된다.
- operator=( ) 함수에 관한 두 가지 다른 중요한 특성이 있다는 것을 주목하라.
- 첫 번째, 이 함수는 참조 매개변수를 사용한다. 이것은 대입문의 우측에 있는 객체가 복사되는 것을 방지한다.
- 앞장에서 보았듯이, 객체가 함수에 전달될 때 복사본이 만들어지고, 그 복사본은 함수가 종료될 때 소멸된다. 이러한 경우에, 복사본을 소멸하기 위해 p를 해제할 소멸자를 호출할 것이다. 그러나 이것은 여전히 인수로서 사용된 객체가 필요로 하는 p이다. 참조 매개변수를 사용하면 이 문제는 해결된다.
- operator=( ) 함수의 두 번째 중요한 특성은 객체가 아닌 참조 매개변수를 반환한다는 것이다. 이것은 참조 매개변수를 사용하는 이유와 같다. 함수가 객체를 반환할 때 임시 객체가 생성되고 반환이 이루어진 후 소멸된다. 그러나 이것은 임시 객체의 소멸자를 호출하여 p를 해제하게 하지만, p는(그것이 가리키는 메모리) 여전히 값을 대입 받는 객체에 의하여 사용된다는 것을 의미한다. 그러므로 참조 매개변수를 반환하면, 임시 객체가 생성되는 것을 방지할 수 있다.

8. 교환 법칙 해결하기

```
#include <iostream>
using namespace std;
class Point {
private:
      int x, y;
public:
      Point(int _x=0, int _y=0):x(_x), y(_y){}
      void ShowPosition();
      Point operator+(int val); //operator+라는 이름의 함수
};
void Point::ShowPosition() {
      cout<<x<<" "<<y<<endl;
}
Point Point::operator+(int val) {
      Point temp(x+val, y+val);
      return temp;
}
int main(void){
      Point p1(1, 2);
      Point p2=p1+10;
     // Point p2 = 10+p1;  ??
      p2.ShowPosition();
      return 0;
}
```

그림: + 연산자에 대한 결합 법칙의 성립

- 10+p; -> 10.operator+(p); ?
- 문제해결: 전역함수에 의한 연산자 오버로딩
- 10+p; -> opreator+(10,p); O.K.

```
Point operator+(int val, Point &p)
{
   return p+val;  // 피연산자의 위치만 변경
}
```

8. 교환 법칙 해결하기

```
#include <iostream>
using namespace std;
class Point {
private:
      int x, y;
public:
      Point(int _x=0, int _y=0):x(_x), y(_y){}
      void ShowPosition();
      Point operator+(int val); //operator+라는 이름의 함수
      friend Point operator+(int val, const Point& p);
};
void Point::ShowPosition() {
      cout<<x<<" "<<y<<endl;
}
Point Point::operator+(int val) {
      Point temp(x+val, y+val);
      return temp;
}
Point operator+(int val, Point& p){
      return p+val;
}
int main(void){
      Point p1(1, 2);
      Point p2=p1+3;
      p2.ShowPosition();

      Point p3=3+p2;
      p3.ShowPosition();

      return 0;
}
```

- 또는

```
Point operator+(int val, Point &p)
{
   Point temp(p.x+val, p.y+val);
   return temp;
}
```


9. cout, cin 그리고 endl에 대한 이해

```
#include<stdio.h>

namespace mystd  //mystd라는 이름공간 시작.
{
      char* endl="\n";
      class ostream // 클래스 ostream 정의
      {
      public:
            void operator<<(char * str) {
                  printf("%s", str);
            }
            void operator<<(int i) {
                  printf("%d", i);
            }
            void operator<<(double i) {
                  printf("%e", i);
            }
      };

      ostream cout;  //ostream 객체 생성
} // mystd 이름공간 끝.
```

```
using namespace mystd;


int main()
{
cout<<"Hello World \n";
cout<<3.14;
cout<<endl;
cout<<1;
cout<<endl;

//   cout<<"Hello World"<<100<<endl; // Error

                                          return 0;
}
```

9. cout, cin 그리고 endl에 대한 이해

- cout<<"Hello World"<<100<<endl;

-> ((cout <<“Hello World”)<<100)<<endl;

((cout <<“Hello World”) << 100)<<endl;

↓

연산후 cout 객체 리턴

(cout << 100) <<endl;

↓

연산후 cout 객체 리턴

↓

cout <<endl;

9. cout, cin 그리고 endl에 대한 이해

```
#include<stdio.h>

namespace mystd   //mystd라는 이름공간 시작
{
  char* endl="\n";
  class ostream      // 클래스 ostream 정의
  {
  public:
    ostream& operator<<(char * str) {
      printf("%s", str);
      return *this;
    }
    ostream& operator<<(int i) {
      printf("%d", i);
      return *this;
    }
    ostream& operator<<(double i) {
      printf("%e", i);
      return *this;
    }
  };

  ostream cout;      //ostream 객체 생성
}                    // mystd 이름공간 끝
```

```
using namespace mystd;

int main()
{
  cout<<"Hello World"<<endl<<3.14<<endl;
  return 0;
}
```

<<, >> 연산자 오버로딩

- cout 과 cin 은 각각 ostream과 istream 클래스의 객체이다.
- 이 둘은 모두 표준 이름공간 std 안에 선언되어 있는 클래스들이다.
- “Point 객체의 멤버변수를 출력하는 방법은 기본 자료형 데이터를 출력하는 방법과 다르다. Point 객체가 지니고 있는 데이터를 다음과 같이 출력할 수는 없는가?

```
int main(void)
{
   point p(1,2);
   cout << p;   // [1, 2] 가 출력
}
```


<<, >> 연산자 오버로딩

- <<, >> 연산자의 오버로딩
  - Point 객체를 기반으로 하는 <<, >> 입 출력 연산

1. 멤버함수에 의한 오버로딩 : No. 이유는?
   cout<<p → cout.operator<<(p); // x (이유는?)
2. 전역함수에 의한 오버로딩 : O.K.
   cout<<p → operator<<(cout, p); // (o)

```
ostream& operator<<(ostream& os, const Point& p)
```

<<, >> 연산자 오버로딩

```
#include <iostream>

using std::cout;
using std::endl;

using std::ostream;

class Point {
private:
	int x, y;
public:
	Point(int _x=0, int _y=0):x(_x), y(_y){}
	friend ostream& operator<<(ostream& os, const Point& p);
};

ostream& operator<<(ostream& os, const Point& p)
{
	os<<"["<<p.x<<", "<<p.y<<"]"<<endl;
	return os;
}

int main(void)
{
	Point p(1, 3);
	cout<<p;	// operator <<(cout, p); 로 해석된다.

	return 0;
}
```


실습문제

- 앞에서 정의된 Point 클래스를 이용한 데이터의 입력이 가능하도록 연산자를 오버로딩하여라. cin 은 이름공간 std안에 선언되어 있는 istream 클래스의 객체이다.

- Main 함수의 예

```
int main(void){
 Point p;
 cout << p;
 cout << "x, y 입력: " ;
 cin >> p; // operator >> (cin, p);
 cout << p;

 return 0;
}
```

질문 & 답변

Thank You !

수고하셨습니다.