10장 연산자 중복과 프렌드 함수

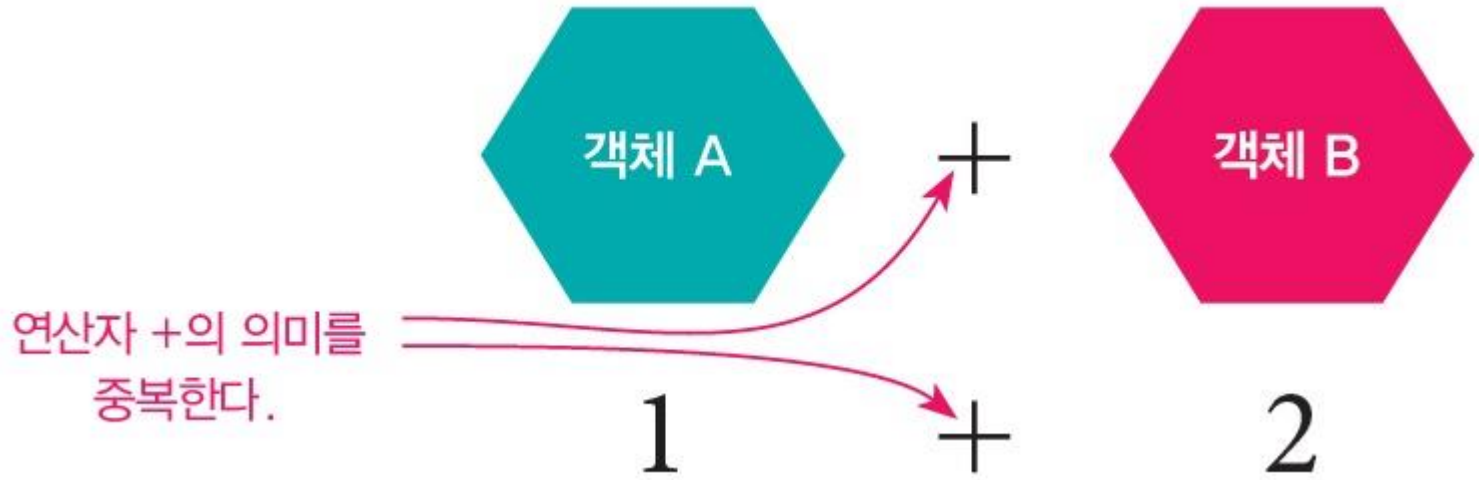
2020. 10. 29

순천향대학교 컴퓨터 공학과

내용

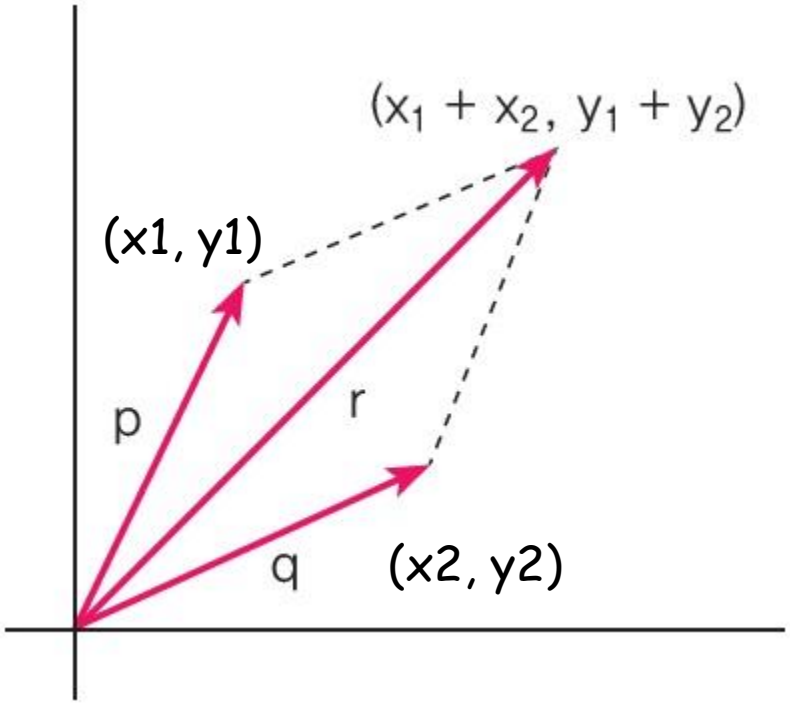
- 연산자 중복
- 프렌드 함수

연산자 중복



연산자 중복의 예

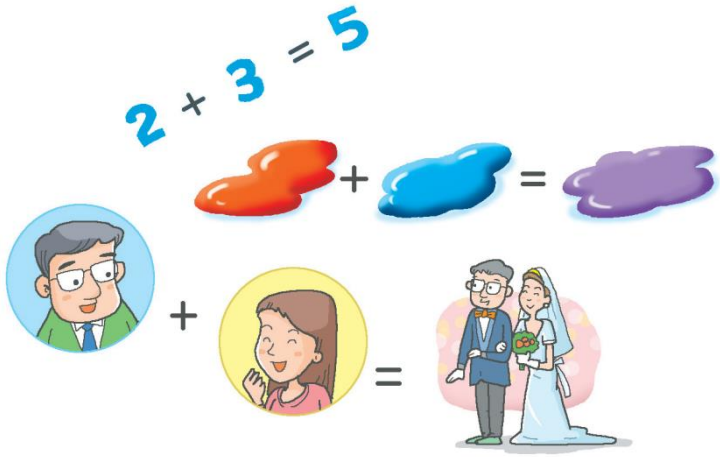
벡터의 경우, + 연산자로 표시하는 것이 더 직관적임




```
MyVector v1, v2, v3;
cout << (v1 + v2 + v3); // 연산자 중복
cout << add(v1, add(v2, v3));// 함수 사용
```

연산자 중복

- 일상 생활에서의 기호 사용
  - + 기호를 숫자와 물체에 적용, 중복 사용
    - 숫자 더하기 : 2 + 3 = 5
    - 색 혼합 : 빨강 + 파랑 = 보라
    - 생활 : 남자 + 여자 = 결혼



  - 간결한 의미 전달
  - 다형성

- C++는 연산자 중복 지원
  - C++에서 제공하는 연산자에 새로운 의미 정의
  - 높은 프로그램 가독성

연산자 중복 예 : + 연산자

```
int a=2, b=3, c;
c = a + b;
```

```
string a="C", c;
c = a + "++"; // 두 개의 문자열 합치기
```

```
Color a(BLUE), b(RED), c;
c = a + b; // 두 가지 색상을 섞어서 생성되는 새로운 Color 객체
```

```
SortedArray a(2,5,9), b(3,7,10), c;
c = a + b; // 정렬된 두 배열을 정렬 유지하면서 통합
```

연산자 중복의 특징

- C++에서 제공된 연산자만 중복 가능
  - 3%%5 // 컴파일 오류
  - 6## 7 // 컴파일 오류

- 연산자 중복은 함수 형태로 구현 - 연산자 함수(operator function)로 정의
  - 피 연산자 타입이 다른 새로운 연산 정의
  - 피연산자의 개수를 바꿀 수 없음
  - 연산자의 우선 순위 변경 안됨

중복 가능한 연산자

- 모든 연산자가 중복 가능하지 않음
- 중복 가능한 연산자

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

중복할 수 없는 연산자

연산자	설명
::	범위 지정 연산자
.	멤버 선택 연산자
.*	멤버 포인터 연산자
?:	조건 연산자

연산자 중복 예: string 클래스

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

int main() {
        string s1 = "Rogue One: ";
        string s2 = "A Star Wars Story";

        string s3;
        s3 = s1 + s2;

        cout << "s1=" << s1 << endl;
        cout << "s2=" << s2 << endl;
        cout << "s1+s2= " << s3 << endl;
        cout << "s1==s2 " << boolalpha << (s1 == s2) << endl;

        return 0;
}
```

연산자 중복 정의 방법

- 연산자 중복은 연산자 함수로 정의
- 연산자 함수 구현 방법 2가지
  1. 클래스의 멤버 함수로 구현
  2. 외부 함수로 구현하고 클래스에 프렌드 함수로 선언
- 연산자 함수 형식:

  리턴타입 **operator**연산자(*매개변수리스트*);

  - Ex. Color operator+(Color opnd2);
    bool operator==(Color opnd2);

예제

- 연산자 함수 작성이 필요한 코드 사례

```
Color a(BLUE), b(RED), c;

c = a + b;  // a와 b를 더하기 위한 + 연산자 작성 필요
if(a == b) { // a와 b를 비교하기 위한 == 연산자 작성 필요
            ...
}
```

연산자 함수 작성 방법 (1)

- 클래스의 멤버 함수로 작성

```
class Color {
   ...
   Color operator+ (Color op2);
   bool operator== (Color op2);
};
```

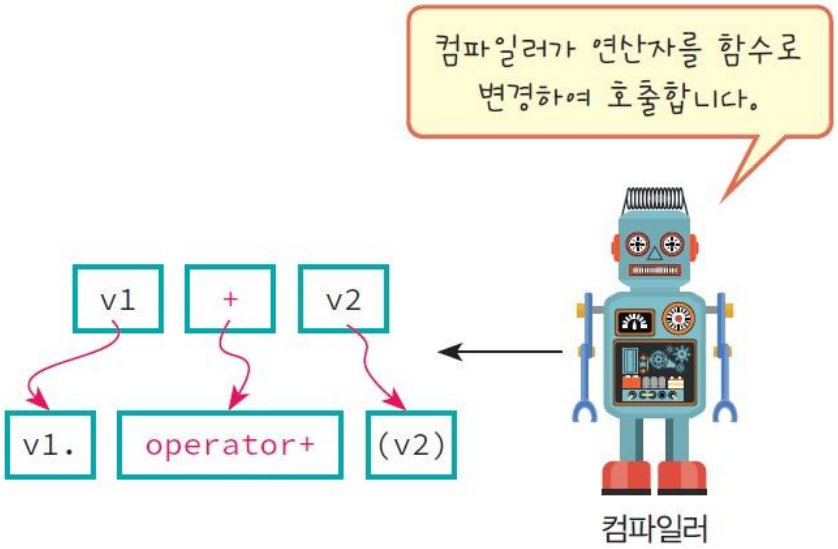


그림 10.1 멤버 함수로 연산자 중복

연산자 함수 작성 방법 (2)

- 외부 함수로 구현되고, 클래스에 프렌드로 선언

```
Color operator + (Color op1, Color op2); // 외부 함수
bool operator == (Color op1, Color op2); // 외부 함수

class Color {
// 연산자 함수를 Color의 프렌드 함수로 지정

  friend Color operator+ (Color op1, Color op2);
  friend bool operator== (Color op1, Color op2);
  …
};
```

+ 연산자 중복

```
class MyVector
{
private:
        double x, y;
public:
        MyVector(double x = 0.0, double y = 0.0) : x{x}, y{y} {}
        string toString() {
                return "("+to_string(x) +", "+to_string(y)+")" ;
        }
        MyVector operator+(const MyVector& v2); // 멤버 함수로 구현
};
```

+ 연산자 중복 (2)

```
MyVector MyVector::operator+(const MyVector& v2) {
  MyVector v;

  v.x = this->x + v2.x;
  v.y = this->y + v2.y;

  return v;
}

int main() {
  MyVector v1(1.0, 2.0), v2(3.0, 4.0);
  MyVector v3 = v1 + v2;

  cout<<v1.toString()<<"+"<<v2.toString()<<"="<<v3.toString()<<endl;

  return 0;
}
```

== 연산자의 중복

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

class Time
{
        int hour, min, sec;
public:
        Time(int h=0, int m=0, int s=0) : hour(h), min(m), sec(s) { }

        bool operator == (Time &t2) { // 멤버 함수로서 연산자 중복
          return (hour == t2.hour && min == t2.min && sec == t2.sec);
        }

        bool operator != (Time &t2) {
          return !(*this == t2);
        }
};
```

프렌드 메카니즘

친구란?

내 가족의 일원은 아니지만 내 가족과 동일한 권한을 인정받은 사람

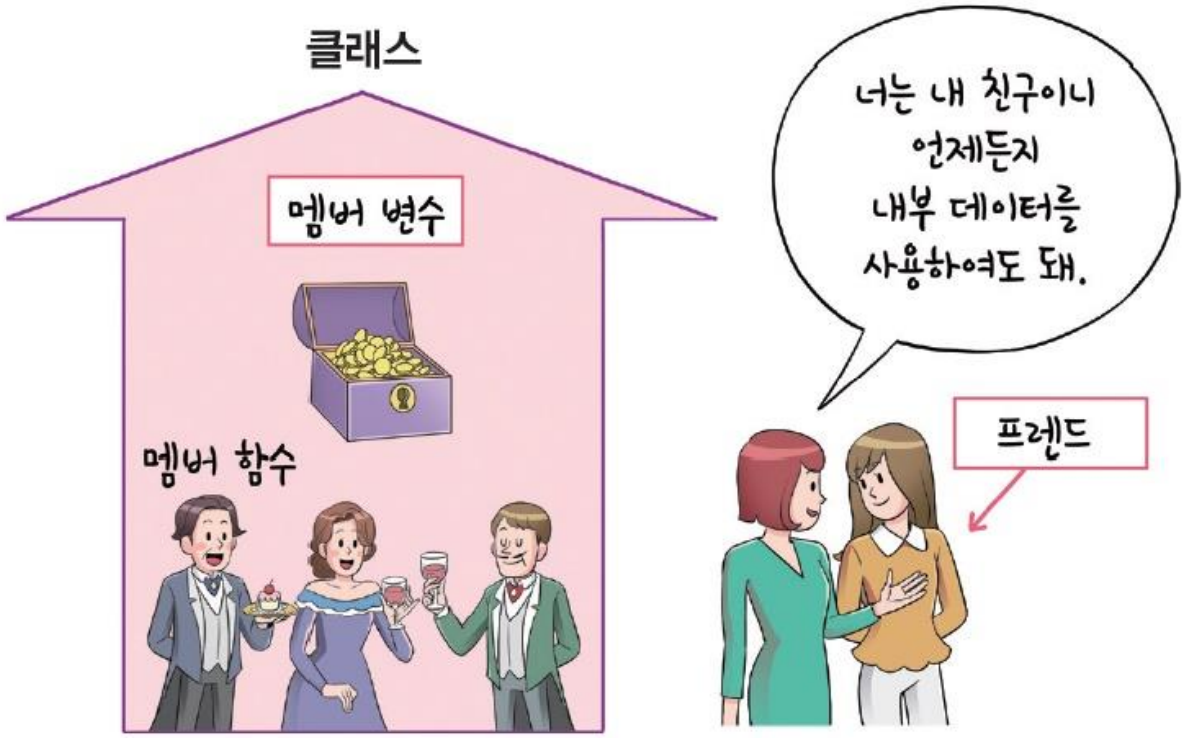


그림 10.2 프렌드의 개념

C++ 프렌드 함수

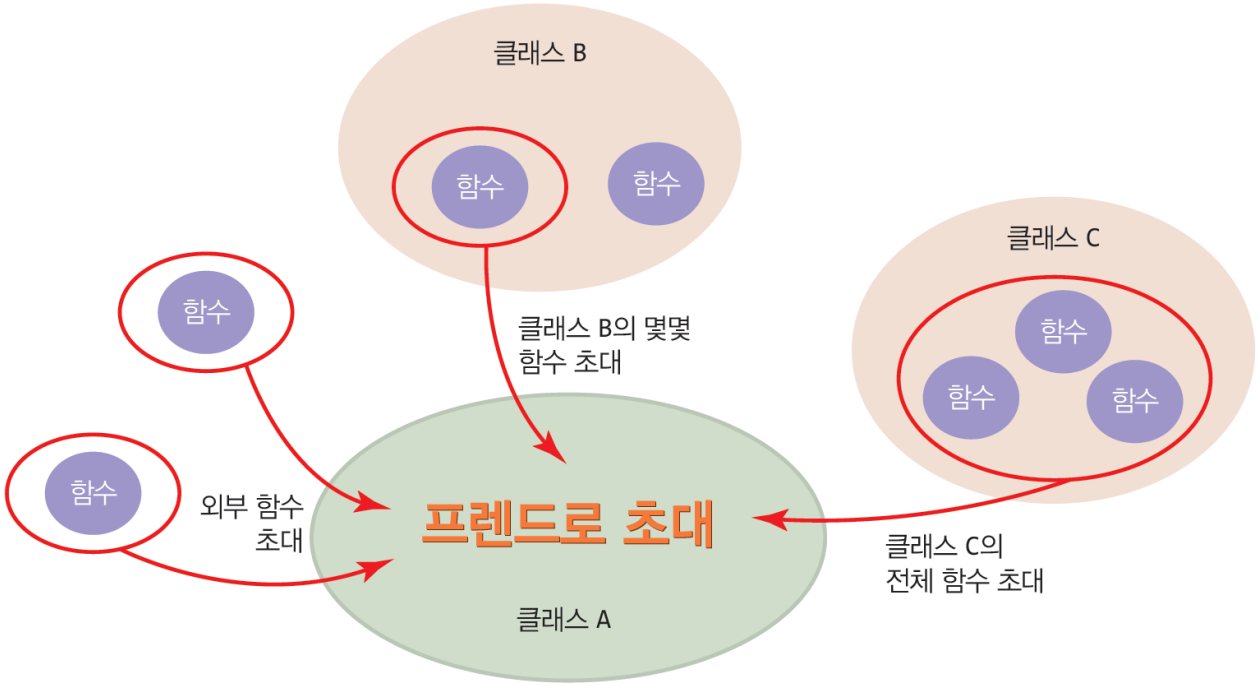
- 프렌드 함수의 특징
  - 클래스의 멤버가 아닌 외부 함수 (전역 함수, 다른 클래스의 멤버 함수)
  - friend 키워드로 클래스 내에 선언된 함수
  - 클래스의 모든 멤버를 접근할 수 있는 권한 부여

- 프렌드 선언의 필요성
  - 클래스의 멤버로 선언하기에는 무리가 있고, 클래스의 모든 멤버를 자유롭게 접근할 수 있는 일부 외부 함수 작성 시

항목	세상의 친구	프렌드 함수
존재	가족이 아님. 외부인	클래스 외부에 작성된 함수. 멤버가 아님
자격	가족의 구성원으로 인정받음. 가족의 모든 살림살이에 접근 허용	클래스의 멤버 자격 부여. 클래스의 모든 멤버에 대해 접근 가능
선언	친구라고 소개	클래스 내에 friend 키워드로 선언
개수	친구의 명수에 제한 없음	프렌드 함수 개수에 제한 없음

프렌드로 초대하는 3 가지 유형

- 프렌드 함수가 되는 3 가지 경우
  - 외부 함수 : 클래스 외부에 선언된 외부 함수
  - 다른 클래스의 멤버 함수 : 다른 클래스의 특정 멤버 함수
  - 다른 클래스 전체 : 다른 클래스의 모든 멤버 함수



프렌드 선언 경우

1. 외부 함수 **equals()**를 **Rect** 클래스에 프렌드로 선언

```
class Rect { // Rect 클래스 선언
   ...
   friend bool equals(Rect r, Rect s);
};
```

2. **RectManager** 클래스의 **equals()** 멤버 함수를 **Rect** 클래스에 프렌드로 선언

```
class Rect {
   ..............
   friend bool RectManager::equals(Rect r, Rect s);
};
```

3. **RectManager** 클래스의 모든 멤버 함수를 **Rect** 클래스에 프렌드로 선언

```
class Rect {
   ..............
   friend RectManager;
};
```

예제

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

class Box {
        double length, width, height;
public:
        Box(double l, double w, double h) : length{l}, width{w}, height{h} { }
        friend void printBox(Box box);
};

void printBox(Box box) {
        cout << "Box( " << box.length << ", " << box.width
        << ", " << box.height<<") " << endl;
}
```

```
int main() {
        Box box(10, 20, 30);

        printBox(box);
        return 0;
}
```

예제

```
class A {
public:
        friend class B;     // B는 A의 프렌드
        A(string s = "") : secret{s} { }
private:
        string secret;      // B의 모든 멤버 함수에서 접근 가능
};

class B {
public:
        B() {   }
        void print(A obj) {
                cout << obj.secret << endl;
        }
};
```

```
int main() {
  A a("이것은 기밀 정보입니다.");
  B b;
  b.print(a);

  return 0;
}
```

예제

```
class Rect {
   int width, height;
public:
   Rect(int width, int height)  {  this->width = width; this->height = height;  }
   friend bool equals(Rect r, Rect s);
};

bool equals(Rect r, Rect s) {
   if(r.width == s.width && r.height == s.height) return true;
   else return false;
}

int main() {
   Rect a(3,4), b(4,5);

   if(equals(a, b)) cout << "equal" << endl;
   else cout << "not equal" << endl;
}
```

```
class Rect; // 전방 선언: Rect가 정의되기 전에 참조되는 것을 허용
class RectManager {
public:
  bool equals(Rect r, Rect s);
};

class Rect {
  int width, height;
public:
  Rect(int width, int height) { this->width = width; this->height = height; }
  friend bool RectManager::equals(Rect r, Rect s); // 다른 클래스에 속한 함수
};

bool RectManager::equals(Rect r, Rect s) {
  if(r.width == s.width && r.height == s.height) return true;
  else return false;
}
int main() {
  Rect a(3,4), b(3,4);
  RectManager man;

  if(man.equals(a, b)) cout << "equal" << endl;
  else cout << "not equal" << endl;
}
```

```
int main() {
  Rect a(3,4), b(5,6);
  RectManager man;

  man.copy(b, a);
  if(man.equals(a, b)) cout << "equal" << endl;
  else cout << "not equal" << endl;
}
```

```
class Rect;

class RectManager {
public:
  bool equals(Rect r, Rect s);
  void copy(Rect& dest, Rect& src);
};

class Rect {
  int width, height;
public:
  Rect(int width, int height)  { this->width = width; this->height = height; }
  friend RectManager; // 다른 클래스를 프랜드로 선언
};

bool RectManager::equals(Rect r, Rect s) {
  if(r.width == s.width && r.height == s.height) return true;
  else return false;
}

void RectManager::copy(Rect& dest, Rect& src) { // src를 dest에 복사
  dest.width = src.width;  dest.height = src.height;
}
```

객체 비교 함수 작성 (1)

```
class Date
{
  int year, month, day;
public:
  Date(int y=0, int m=0, int d=0) : year(y), month(m), day(d) {   }
  bool equals(Date obj) {
    return year == obj.year && month == obj.month && day == obj.day;
  }
};
```

```
int main() {
  Date d1(1960, 5, 23), d2(2002, 7, 23);
  if( d1.equals(d2) == true )
            cout << “Equal" << endl;
 else
            cout << "Different" << endl;
}
```

객체 비교 함수 작성 (2)

```
class Date {
        int year, month, day;
public:
        Date(int y=0, int m=0, int d=0) : year(y), month(m), day(d) {   }
        friend bool equals(Date d1, Date d2);
};

bool equals(Date d1, Date d2) {
  return d1.year == d2.year && d1.month == d2.month
        && d1.day == d2.day;
}
```

```
int main() {
  Date d1(1960, 5, 23), d2(2002, 7, 23);
  if( equals(d1, d2) == true )
        cout << “Equal" << endl;
 else
        cout << "Different" << endl;
}
```

프랜드 함수 용도

- 2개의 객체에 대한 연산 표현
- 연산자 함수를 외부 함수로 정의할 때

예제

```
class Complex {
private:
        double re, im;
public:
        friend Complex add(Complex, Complex);
        Complex(double r=0.0, double i=0.0) { re = r; im = i; }
        void print() {
                cout << re << " + " << im << "i" << endl;
        }
};

Complex add(Complex a1, Complex a2)
{
  return Complex(a1.re + a2.re, a1.im + a2.im);
}
```

```
int main() {
  Complex c1(1, 2), c2(3, 4);
  Complex c3 = add(c1, c2);
  c3.print();
  return 0;
}
```

예제: Power 클래스

- 다음 Power 클래스를 생각해보자.

```
class Power { // 에너지를 표현하는 파워 클래스
  int kick;      // 발로 차는 힘
  int punch; // 주먹으로 치는 힘
public:
  Power(int kick=0, int punch=0) {
    this->kick = kick;
    this->punch = punch;
   }
};
```

Power 객체를 더하는 + 연산자

- 2개의 Power 객체를 더하는 경우 고려

```
int main() {
   Power a(3,5), b(4,6), c;

   c = a + b; // power 객체를 더한다
   a.show();
   b.show();
   c.show();
}
```

```
class Power {
   int kick;
   int punch;
public:
   Power(int kick=0, int punch=0) {
      this->kick = kick;
      this->punch = punch;
    }
};
```

2 + a 덧셈 연산은?

- 다음 코드는 올바른가?

```
Power a(3,4), b;

b = 2 + a; // + 연산자 중복을 어떻게 정의할 것인가?
```

Power 객체를 더하는 + 연산자 중복

```
class Power {
   int kick;
   int punch;
public:
   Power(int kick=0, int punch=0) {
      this->kick = kick; this->punch = punch;
   }
   void show();
   Power operator+ (Power op2); // + 연산자 함수 선언
};

void Power::show() {
   cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power Power::operator+(Power op2) {
   Power tmp;
   tmp.kick = this->kick + op2.kick;
   tmp.punch = this->punch + op2.punch;
   return tmp;
}
```

Power 객체를 더하는 + 연산자 중복 (2)

```
class Power {
   int kick;
   int punch;
public:
   Power(int kick=0, int punch=0) {
      this->kick = kick; this->punch = punch;
   }
   void show();
   friend Power operator+(Power op1, Power op2);
};

void Power::show() {
   cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power operator+(Power op1, Power op2) {
   Power tmp;
   tmp.kick = op1.kick + op2.kick;
   tmp.punch = op1.punch + op2.punch;
   return tmp;
}
```

2 + a 덧셈 연산은? => 프랜드 함수 이용

```
class Power {
   int kick;
   int punch;
public:
   Power(int kick=0, int punch=0) {
      this->kick = kick; this->punch = punch;
   }
   void show();
   friend Power operator+(int op1, Power op2);
};

void Power::show() {
   cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power operator+(int op1, Power op2) {
   Power tmp;
   tmp.kick = op1 + op2.kick;
   tmp.punch = op1 + op2.punch;
   return tmp;
}
```

```
int main() {
   Power a(3,5), b;
   a.show();
   b.show();
   b = 2 + a;
   a.show();
   b.show();
}
```