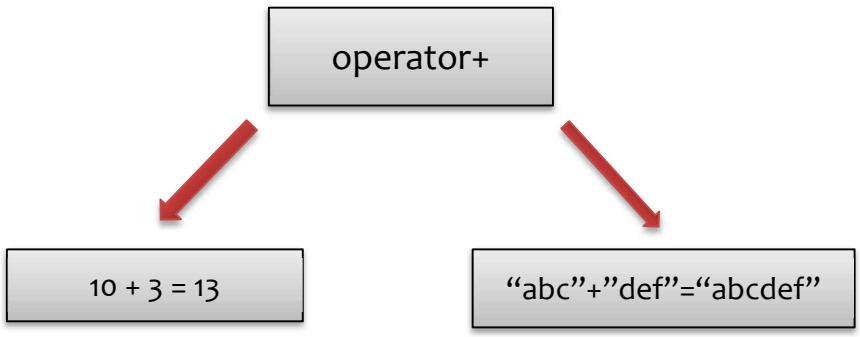
제3장 연산자 오버로딩

클래스의 성격에 따라 필요한 연산기능이 있다면 그에 맞게 동작하도록 기본연산자의 기능에 새로운 기능을 수행할 수 있도록 해주는 것을 연산자 오버로딩이라 한다.



operator overloading

◈ 연산자 오버로딩(operator overloading) 함수

“operator연산자기호”의 형식으로 정의되는 함수
클래스 타입만 연산자 오버로딩이 지원됨 (기본 데이터 타입은 불가능)
리턴 타입은 연산자 오버로딩 함수의 종류에 따라 다름(클래스 타입, bool, istream 등)

◈ 연산자 오버로딩(operator overloading) 함수의 구현 방식

방법 1 : 클래스의 멤버 함수로 정의하는 방법
방법 2 : 전역 함수(extern function)로 정의하는 방법

▶ **멤버함수 형태로만 중복정의 가능한 연산자**

연산자	중복함수이름
=	대입연산자
()	함수호출연산자
[]	배열원소참조연산자
->	멤버참조연산자

▶ **연산자 오버로딩 불가능 연산자**

연산자	중복함수이름
::	범위지정연산자
.	멤버선택연산자
?:	3 항 연산자
.*	멤버 포인터연산자

1. 대입연산자 함수

- 객체들끼리의 대입을 자유롭게 수행할 수 있도록 지원되는 멤버 함수
- 대입연산자함수는 객체에 대입연산자를 사용할 때 자동으로 호출되어 멤버들을 복사해서 다른 객체에 대입시키는 기능을 수행한다.
- 사용자가 정의하지 않을 시 컴파일러는 대입연산자 멤버함수를 기본적으로 추가한다.
  (이 때 멤버 대 멤버가 그대로 복사되는 형태로 수행 됨)

기본적으로 추가되는 대입연산자 멤버함수는 모든 데이터가 객체 안에 저장되어 있을 때는 문제없이 수행된다. 그러나 포인터멤버가 동적할당 영역을 연결하고 있다면 포인터 멤버의 값만 복사되므로 두 객체가 같은 동적할당 기억공간을 연결하는 문제가 발생한다.

[예제3-1] 두 객체를 교환하는 프로그램

```
1:  #include <iostream.h>
2:  #include <string.h>
3:
4:  // 클래스 선언
5:  class MoneyBox{
6:  private:
7:          char *name;
8:          int sum;
9:  public:
10:         MoneyBox(const char *np="아무개", int m=0);   // 생성자
11:         MoneyBox(const MoneyBox &br);   // 복사 생성자
12:         ~MoneyBox();   // 소멸자
13:         MoneyBox &operator=(const MoneyBox &br);   // 대입연산자
14:         void save(int m);
15:         void use(int m);
16:         void prn();
17: };
18:
19: // 멤버함수 정의
20: MoneyBox::MoneyBox(const char *np, int m)
21: {
22:         name=new char[strlen(np)+1];
23:         strcpy(name, np);
24:         sum=m;
25: }
26:
27: MoneyBox::MoneyBox(const MoneyBox &br)
28: {
29:         name=new char[strlen(br.name)+1];
30:         strcpy(name, br.name);
31:         sum=br.sum;
32: }
33:
34: MoneyBox::~MoneyBox()
35: {
36:         delete [] name;
37: }
38:
39: MoneyBox &MoneyBox::operator=(const MoneyBox &br)
40: {
41:         if(this==&br) return *this;
42:         delete [] name;
43:         name=new char[strlen(br.name)+1];
44:         strcpy(name, br.name);
45:         sum=br.sum;
46:         return *this;
47: }
48:
```

```
49: void MoneyBox::save(int m)
50: {
51:         sum+=m;
52: }
53:
54: void MoneyBox::use(int m)
55: {
56:         sum-=m;
57: }
58:
59: void MoneyBox::prn()
60: {
61:         cout << "이름 : " << name << endl;
62:         cout << "잔액 : " << sum << endl;
63: }
64:
65: // 메인 함수
66: int main()
67: {
68:         MoneyBox a("철이", 100), b("메텔", 500), temp;
69:         a.prn();
70:         b.prn();
71:         temp=a;
72:         a=b;
73:         b=temp;
74:         a.prn();
75:         b.prn();
76:
77:         a=b=temp;   // 연속사용 가능
78:         a=a;   // 자신에 대입 가능
79:
80:         return 0;
81: }
```

실행결과

```
이름 : 철이
잔액 : 100
이름 : 메텔
잔액 : 500
이름 : 메텔
잔액 : 500
이름 : 철이
잔액 : 100
```

2. 사용자 정의 연산자 멤버함수

대입연산자 멤버함수는 별도로 정의하지 않으면 암시적인 버전이 자동으로 추가된다. 반면에 다른 연산자 멤버함수는 클래스의 특징에 맞게 명시적으로 정의해야 사용이 가능하다.

[예제3-2] 연산자 오버로딩을 사용한 객체의 연산

```
1:   #include <iostream>
2:   using namespace  std;
3:   class Time{
4:   private:
5:        int hour;  // 시간을 저장할 데이터멤버
6:        int min;  // 분을 저장할 데이터멤버
7:   public:
8:        Time(int h=0, int m=0);  // 생성자
9:        Time operator+(const Time &br);  // 덧셈연산자 멤버함수
10:       Time operator-();  // Unary minus 연산자 멤버함수
11:       Time &operator++();  // 전위표기 증가연산자
12:       Time operator++(int);  // 후위표기 증가연산자
13:       void show();  // 시간을 출력하는 멤버함수
14:  };
15:
16:  Time::Time(int h, int m)
17:  {
18:       hour=h;
19:       min=m;
20:  }
21:
22:  Time Time::operator+(const Time &br)
23:  {
24:       int h, m;  // 시간, 분을 임시로 저장할 변수
25:       h=hour+br.hour;  // 시간의 합
26:       m=min+br.min;  // 분의 합
27:       if(m>=60){  // 더한 값이 60분 이상이면
28:            h+=m/60;  // 시간에 추가
29:            m=m%60;  // 나머지 분 계산
30:       }
31:       Time res(h, m);  // 임시객체 생성
32:       return res;  // 임시객체를 리턴한다.
33:  }
34:
35:  Time Time::operator-()  // 매개변수가 없다.
36:  {
37:        Time temp(-hour, -min);  // 부호를 바꾸어 임시객체 생성
```

```
38:         return temp;   // 임시객체 리턴
39:   }
40:
41:   Time &Time::operator++()   // 전위표기 증가연산자
42:   {
43:         min++;
44:         if(min==60){
45:               hour++;
46:               min=0;
47:         }
48:         return *this;   // 증가된 객체를 리턴한다.
49:   }
50:
51:   Time Time::operator++(int)   // 후위표기 증가연산자
52:   {
53:       Time temp(hour, min);   // 증가되기 전의 객체를 임시 보관한다.
54:       min++;
55:       if(min==60){
56:             hour++;
57:             min=0;
58:       }
59:       return temp;    // 증가되기 전의 객체를 리턴한다.
60:   }
61:
62:   void Time::show()
63:   {
64:       cout << hour << "시간 " << min << "분" << endl;
65:   }
66:
67:   void func(Time &br);
68:
69:   int main()
70:   {
71:       Time a(5, 50), b(3, 30), c;
72:
73:       c=-b+a;   // c.operator=(b.operator-().operator+(a))
74:       a.show();   // 5시간 50분
75:       b.show();   // 3시간 30분
76:       c.show();   // 2시간 20분
77:
78:       c=++a+b;
79:       a.show();   // 5시간 51분
80:       b.show();   // 3시간 30분
81:       c.show();   // 9시간 21분
82:
83:       c=a+b++;
84:       a.show();   // 5시간 51분
```

```
85:       b.show();   // 3시간 31분
86:       c.show();   // 9시간 21분
87:
88:       func(++a);
89:       a.show();   // 5시간 53분
90:
91:       return 0;
92:   }
93:
94:   void func(Time &br)
95:   {
96:       br++;
97:   }
```

3. 다른 자료형과 연산할 때 오버로딩 방법

• 연산자 오버로딩은 같은 객체뿐만 아니라 기본자료형이나 다른 객체와의 연산도 가능하다.
Time객체와 정수와의 곱을 시간과 분을 모두 정수배하는 연산으로 정의한다면 다음과 같이 곱하기 연산자를 오버로딩한다.

```
Time Time::operator*(int n)  // 곱하는 정수값을 매개변수로 받는다.
{
    int h, m;
    h=hour*n;
    m=min*n;
    if(m>=60){
        h+=m/60;
        m=m%60;
    }
    return Time(h, m);
}

Time a(2, 30), b;
b=a*3;  // a.operator*(3) 호출, a의 시간과 분에 각각 3을 곱한다.
b.show();  // 7시간 30분
```

• 연산자 멤버함수를 호출하는 객체가 반드시 연산자 왼쪽에 오도록 해야 한다.
기본자료형이 아닌 객체에 연산을 수행할 때는 왼쪽 피연산자 객체에 연산자 오버로딩이 되어 있는지 살펴보고 있으면 그 객체를 통해서 호출하기 때문이다.
따라서 3*a와 같은 연산은 불가능하고 결국 곱셈의 교환법칙이 성립하지 않게 된다.
이 문제를 해결하기 위해서는 멤버함수가 아닌 외부함수로 연산자 오버로딩을 하는 방법이 있다.

• 두 자료형을 전달인자로 받을 수 있는 연산자 오버로딩 되어 있는 외부함수를 찾아본다.
따라서 int형과 Time형의 객체를 모두 전달인자로 받는 operator* 외부함수를 정의하고 함수 안에서 두 피연산자의 위치를 바꾸면 된다.

```
Time operator*(int n, Time &a)  // 클래스 범위가 없으므로 일반함수
{
    return a*n;  // 전달인자로 받은 값의 위치를 바꾸어 연산한다.
}
```

4. 형변환을 수행하는 오버로디드 생성자

매개변수가 하나인 생성자는 매개변수의 형을 클래스형으로 자동 변환하는 형변환함수의 기능을 수행하고 형변환이 필요한 곳에서 자동으로 호출되어 임시 객체를 생성한다.

```
// 대입연산을 수행하는 경우
Time a;
a=3.5;   // a.operator=(Time(3.5));
a=3;   // 3이 3.0으로 자동형변환된 후에 a.operator=(Time(3.0));

// 함수의 전달인자나 리턴값으로 실수값이 사용된 경우
Time func(Time t)
{
    ...
    return 3.5;   // Time(3.5)호출되어 임시객체를 생성한 후에 리턴한다.
}

func(3.5);   // Time(3.5)가 호출되며 임시객체를 생성한 후에 전달한다.
```

▶ 오버로디드 생성자의 형변환 기능 제한하기

explicit를 사용하면 생성자가 호출되어어 자동으로 형변환하는 기능을 제한할 수 있다.
물론 필요하다면 명시적으로 호출하여 형변환하는 것은 가능하다.

```
// 클래스에 선언할 때 앞에 explicit를 붙인다.
explicit Time(double t);

Time a;
a=3.5;   // 에러! 3.5를 Time클래스형으로 변환할 수 없다.
a=Time(3.5);   // 생성자함수의 명시적 호출
a=(Time)3.5;   // 형변환 연산자의 형식으로도 사용할 수 있다.
```

5. 형변환 연산자 함수

하나의 매개변수를 갖는 생성자는 다른 자료형을 클래스형으로 변환하지만 그 반대의 경우는 불가능하다. 예를 들어, 실수값과 Time클래스의 객체를 더하는 연산은 충분히 의미가 있다. 그러나 연산이 가능하게 하려면 연산자를 오버로딩한 멤버함수를 만들거나 프렌드함수를 만들어야 한다.

```
Time a(3.5);
a+1.5;   // a.operator+(1.5), operator+(double) 멤버함수로 가능하다.
1.5+a;   // operator+(double, Time) 프렌드 함수가 있어야 한다.
```

만약 a가 double형으로 변환된다면 그 위치에 상관없이 연산이 가능할 것이다
형변환함수는 'operator'에 변환을 원하는 자료형을 붙여서 이름을 만들며 매개변수는 없고 리턴형은 사용하지 않는다.

```
// Time형을 double형으로 변환하는 형변환함수의 선언
operator double() const;

// 형변환함수의 정의
Time::operator double() const   // 매개변수와 리턴형이 없다.
{
    double temp;
    temp=hour+min/60.0;
    return temp;   // 리턴형은 없지만 값은 리턴해야 한다.
}

Time a(3.5);
double res;

res=1.0+a;   // 형변환함수의 암시적 호출
res=double(a)+1.0;   // 형변환함수의 명시적 호출
res=1.0+(double)a;   // 형변환연산자 형식으로 호출할 수 있다.
```

[예제3-3] 형변환함수를 사용한 프로그램

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

class Time
{
  private:
        int hour;
        int min;
  public:
        Time(int h=0, int m=0);  // 시, 분을 받는 오버로디드 생성자
        Time(double t);  // 시간을 실수값으로 초기화하는 생성자
        operator double() const;  // 형변환함수
        void show(); // 멤버 출력함수
};

Time::Time(int h, int m)
{
        cout << "int, int  생성자 호출..." << endl;
        hour=h;
        min=m;
}

Time::Time(double t)
{
        cout << "double 생성자 호출..." << endl;
        hour=int(t);
        min=(int)((t-hour)*60.0);
}

Time::operator double() const
{
        cout << "형변환함수 호출..." << endl;
        double temp;
        temp=hour+min/60.0;
        return temp;
}

void Time::show()
{
        cout << this->hour << "시간 " << this->min << "분" << endl;
}


int main()
{
        Time a;
        a=3.5;  // 생성자 함수 호출
        a.show();

        double res;
        res=1.0+a;  // operator double() 형변환 멤버함수 호출
        cout << res << endl;
```

```
        return 0;
}
```

실행결과

```
int, int  생성자 호출...
double 생성자 호출...
3시간 30분
형변환함수 호출...
4.5
```

위의 예제를
explicit Time(double t); 로 수정한 결과는?
또한
explicit Time(int h=0, int m=0); 까지 수정한 결과는?

* 형변환함수는 생성자와는 달리 explicit를 사용할 수 없으므로 일단 정의하면 암시적인 사용을 막을 수 없다. 따라서 형변환함수가 두 개이상 정의되면 선택이 모호한 상황이 발생할 수도 있고 불필요한 곳에서 자동형변환되어 문제를 일으킬 가능성도 있다. 그러므로 형변환함수를 사용하는 것 보다 원하는 자료형을 리턴하도록 멤버함수를 정의하고 객체를 통해 명시적으로 호출하는 것이 바람직하다.

```
double Time::to_double();  // double형을 리턴하는 멤버함수 선언

Time a(3.5);
double res=1.5+a.to_double();  // 멤버함수를 호출하여 double값으로 변환한다.
```

6. 입출력 연산자(<<, >>) 함수

입출력 클래스(ostream, istream)의 객체는 사용자가 임의로 생성할 수없다.
그러므로 <<, >> 오버로딩시에는 주의가 필요하다.

[예제3-4] cin과 cout을 사용한 객체의 입출력

```
1:  #include <iostream>
2:  using namespace  std;
3:  class Time{
4:  private:
5:        int hour;  // 시간을 저장할 데이터멤버
6:        int min;  // 분을 저장할 데이터멤버
7:  public:
8:        Time(int h=0, int m=0);  // 생성자
9:        Time operator*(int);  // 정수와 곱하는 연산자 멤버함수
10:       void operator<<(ostream &os);  // 출력연산자 멤버함수
11:       void operator>>(istream &is);  // 입력연산자 멤버함수
12: };
13:
14: Time::Time(int h, int m)
15: {
16:       hour=h;
17:       min=m;
18: }
19:
20: Time Time::operator*(int n)  // 곱하는 정수값을 매개변수로 받는다.
21: {
22:       int h, m;
23:       h=hour*n;
24:       m=min*n;
25:       if(m>=60){
26:             h+=m/60;
27:             m=m%60;
28:       }
29:       return Time(h, m);
30: }
31:
32: void Time::operator<<(ostream &os)
33: {
34:       os << hour << "시간 " << min << "분" << endl;
35: }
36:
37: void Time::operator>>(istream &is)
```

```
38: {
39:         is >> hour >> min;
40: }
41:
42: Time operator*(int n, Time &a);
43: ostream &operator<<(ostream &os, Time &br);
44: istream &operator>>(istream &is, Time &br);
45:
46: int main()
47: {
48:         Time a, b;
49:         cout << "시간과 분을 입력하세요 : ";
50:         cin >> a;
51:         b=3*a;   // a.operator*(3) 호출, a의 시간과 분에 각각 3을 곱함
52:         cout << a << b;
53:
54:         return 0;
55: }
56:
57: Time operator*(int n, Time &a)   // 3*a를 a*3으로 바꾸는 일반함수
58: {
59:         return a*n;   // 전달인자로 받은 값의 위치를 바꾸어 연산한다.
60: }
61:
62: ostream &operator<<(ostream &os, Time &br)   // 출력 일반함수
63: {
64:         br<<os;
65:         return os;
66: }
67:
68: istream &operator>>(istream &is, Time &br)   // 입력 일반함수
69: {
70:         br>>is;
71:         return is;
72: }
```

실행결과

```
시간과 분을 입력하세요 : 2 30(엔터)
2시간 30분
7시간 30분
```

7. 스마트 포인터의 개념

스마트 포인터란 다른 객체의 포인터 역할을 하는 객체로 동적할당된 객체의 메모리 누수를 예방할 목적으로 사용한다.

주로 -> 연산자를 중복정의 하여 사용하며 스마트 포인터 자체는 객체이므로 소멸자, 대입 연산 등이 가능하다는 장점이 있다.

[예제3-5] 스마트 포인터의 동작 예

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

class Obj
{
public:
        Obj()     { cout << "Obj::Obj()" << endl;}
        ~Obj()    { cout << "Obj::~Obj()" << endl;}

        void foo() const  { cout << "Obj::foo" << endl;}
        void goo() const  { cout << "Obj::goo" << endl;}
};
class SmartPointer
{
        Obj* po;
public:
        SmartPointer( Obj* p) : po(p) { }
        Obj* operator->(){ return po; }
        ~SmartPointer()   { delete po; }
};
int   main()
{
        SmartPointer p( new Obj );
        p->goo();     // (p.operator->())->goo(); 의 의미

        return 0;
}
```

** 이후 이 예제를 템플릿을 적용하여 작성해 보자

8. 프렌드(friend) 관계

클래스는 접근지정자를 사용하여 외부로부터 데이터멤버를 보호하고 항상 멤버함수를 통해 처리되도록 함으로서 데이터의 신뢰성을 높인다. 그러나 때로는 융통성 있게 데이터를 처리하는 것이 더 도움이 될 수 있는데, 이 때 프렌드 관계를 이용한다.

- 외부함수를 프렌드로 설정하는 **프렌드 함수**
- 클래스를 프렌드로 설정하는 **프렌드 클래스**
- 다른 **클래스의 멤버함수를 프렌드로** 설정하는 세 가지 경우가 있다.

▌프렌드 함수

프렌드 함수는 외부함수를 프렌드로 설정하는 것이다.

```
class Time{
private:
    int hour;   // 시간을 저장할 데이터멤버
    int min;   // 분을 저장할 데이터멤버
public:
    Time(int h=0, int m=0);   // 오버로디드 생성자
    friend void show(const Time &br);   // 프렌드 함수 선언
};

Time::Time(int h, int m)
{
    hour=h;
    min=m;
}
void show(const Time &br)   // 클래스 표시가 없는 일반함수 정의
{
    cout << br.hour << "시간 " << br.min << "분" << endl;
}
int main()
{
    Time a(2, 30);
    show(a);
    return 0;
}
```

프렌드 함수를 사용하면 입출력 연산자 오버로딩을 보다 간단하게 구현할 수 있다.

```
class Time{
private:
    int hour;   // 시간을 저장할 데이터멤버
    int min;   // 분을 저장할 데이터멤버
public:
    ...
    friend ostream &operator<<(ostream &os, Time &br);   // 프렌드
};
```

```
ostream &operator<<(ostream &os, Time &br)  // 출력연산자 일반함수
{
    os << br.hour << "시간 " << br.min << "분" << endl;
    return os;
}
int main()
{
    Time a(2, 20), b(3, 30);
    cout << a << b;
    return 0;
}
```

▌프렌드 클래스

프렌드 클래스는 다른 클래스 전체를 friend로 선언하는 것이다.

[예제3-6] 프렌드 관계를 사용한 프로그램

```
1:  #include <iostream>
2:  using namespace std;
3:  #include <string.h>
4:  // MoneyBox 클래스 정의
5:  class MoneyBox{
6:      friend class Fund;  // 프렌드 클래스 선언
7:  private:
8:      char name[20];
9:      int sum;
10: public:
11:     MoneyBox(const char *np="아무개", int m=0);
12:     void save(int m);
13:     void use(int m);
14:     friend ostream &operator<<(ostream &os, MoneyBox &br);
15: };
16:
17: // Fund 클래스 정의
18: class Fund{
19: private:
20:     int sum;
21: public:
22:     Fund(int m=0);
23:     void donate(MoneyBox &br, int m);  // 모금 함수
24:     friend ostream &operator<<(ostream &os, Fund &br);
25: };
26:
27: // MoneyBox 클래스의 멤버함수 정의
28: MoneyBox::MoneyBox(const char *np, int m)
29: {
30:     strcpy(name, np);
31:     sum=m;
32: }
33:
34: void MoneyBox::save(int m)
```

```
35: {
36:         sum+=m;
37: }
38:
39: void MoneyBox::use(int m)
40: {
41:         sum-=m;
42: }
43:
44: // Fund 클래스의 멤버함수 정의
45: Fund::Fund(int m)
46: {
47:         sum=m;
48: }
49:
50: void Fund::donate(MoneyBox &br, int m)
51: {
52:         if(m>br.sum){
53:              cout << "잔액이 부족합니다!" << endl;
54:              return;
55:         }
56:         br.sum-=m;
57:         sum+=m;
58:         cout << br.name << "님은 " << m << "원 기부했다." << endl;
59: }
60:
61: // 출력연산자 프렌드함수 선언
62: ostream &operator<<(ostream &os, MoneyBox &br);
63: ostream &operator<<(ostream &os, Fund &br);
64: // 메인함수
65: int main()
66: {
67:         MoneyBox a("철이", 100), b("메텔", 500);
68:         Fund angel;
69:
70:         cout << "기부 전 잔액..." << endl;
71:         cout << a << b << endl;
72:
73:         angel.donate(a, 50);
74:         angel.donate(b, 200);
75:
76:         cout << "₩n기부 후 잔액..." << endl;
77:         cout << a << b << endl;
78:
79:         cout << angel << endl;
80:
81:         return 0;
82: }
83:
84: // 출력연산자 프렌드함수 정의
85: ostream &operator<<(ostream &os, MoneyBox &br)
86: {
87:         os << "이름 : " << br.name << ", ";
88:         os << "잔액 : " << br.sum << endl;
89:         return os;
```

```
90: }
91:
92: ostream &operator<<(ostream &os, Fund &br)
93: {
94:         cout << "현재까지 적립금 : " << br.sum;
95:         return os;
96: }
```

실행결과

```
기부 전 잔액...
이름 : 철이, 잔액 : 100
이름 : 메텔, 잔액 : 500
철이님은 50원 기부했다.
메텔님은 200원 기부했다.
기부 후 잔액...
이름 : 철이, 잔액 : 50
이름 : 메텔, 잔액 : 300
현재까지 적립금 : 250
```

▌프렌드 멤버함수

프렌드 관계는 클래스의 일부 멤버함수에만 적용할 수도 있다. Fund클래스는 MoneyBox의 데이터에 직접 접근할 필요가 있으므로 MoneyBox클래스에서 프렌드 클래스로 선언했지만 실제로 Fund클래스의 donate멤버함수만이 MoneyBox의 데이터멤버를 필요로 한다. 이 때 MoneyBox클래스는 donate멤버함수만을 프렌드로 설정하여 불필요한 접근을 차단할 수 있다.

```
class MoneyBox{
    friend void Fund::donate(MoneyBox &br, int m); // 프렌드 멤버함수
    ...
};

class Fund{
private:
    int sum;
public:
    Fund(int m=0);
    void donate(MoneyBox &br, int m);   // 모금 함수
    friend ostream &operator<<(ostream &os, Fund &br);
};
```

9. string class

문자열 처리를 목적으로 정의된 라이브러리 클래스

▌string 클래스의 특징 ·

char* 형식의 C언어 문자열 처리 가능 ·
string클래스의 문자열 데이터는 char* 포인터 변수와 new를 이용하여 처리
산술연산, 비교 연산 등 연산자 오버로딩 기능이 있어서 문자열 처리가 쉬움
** 헤더파일의 구분
- C언어 문자열 처리를 위한 헤더파일 #include <cstring>
- C++ string 클래스를 위한 헤더파일 #include <string>

▌string 클래스 멤버함수 예시

(생성자)
- string str : 디폴트 생성자, NULL string값의 string 객체 str 생성
- string str("dream") : "dream"데이터를 가진 string 객체 str 생성
- string str(string obj) : obj string 객체가 가지고 있는 데이터를 가진 string 객체 str생성

(데이터 처리 함수])
- str[i] : i 위치에 있는 문자 리턴
- str.at(i) : i 위치에 있는 문자 리턴
- str.empty() : 데이터가 비어있는지 결과 리턴 (true/false)
- str.length() : 데이터 길이 리턴
- str.insert(pos, str2) : pos 위치에 str2 문자열 데이터를 삽입
- str.erase(pos, length) : pos 위치에서 length 길이 만큼 데이터 삭제
- str.find(str1) : str1 문자열 데이터의 첫 번째 문자가 있는 위치 리턴
- str.find(str1, pos) : pos 위치부터 str1 문자열 데이터의 첫번째 문자가 있는 위치 리턴
- str1 = str2 : str2데이터를 str1데이터에 할당
- str1 += str2 : str1데이터에 str1데이터와 str2데이터를 합하여 할당
- str1 + str2 : str1데이터와 str2데이터를 합친 임시 string 객체 반환
- str1 == str2, str1 != str2, str1 < str2, str1 > str2, str1 <= str2, str1 >= str2
  : str1 데이터와 str2 데이터를 문자 비교 (true/false)

(사용 예)

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
        string temp;
        cout << "여백 없는 문자열 입력 : ";
        cin >> temp;
        cout << "여백 없는 문자열 입력 결과 : " << temp << endl;
        cin.ignore(); // 입력 버퍼에서 개행 문자 삭제
        cout << "여백 있는 문자열 입력 : ";
        getline(cin, temp);
        cout << "여백 있는 문자열 입력 결과 : " << temp << endl;
        return 0;
}
```