



Chapter 11. 객체의 다양한 활용

목차

1. 객체 포인터
2. 객체의 매개변수 전달 방식
3. 정적 멤버변수와 정적 멤버함수
4. 객체 배열
5. 프렌드 함수
6. 객체를 다루기 위한 함수
7. 연산자 오버로딩

학습목표

- 객체 포인터를 선언한 후 이를 이용해서 간접 참조한다.
- 자신의 인스턴스를 지칭하는 this에 대해서 학습한다.
- private 멤버를 클래스 외부에서 사용하기 위한 프렌드 함수를 학습한다.
- 객체를 매개변수로 하는 다양한 함수를 정의한다.
- 사용자가 정의한 클래스를 다루기 위한 연산자를 오버로딩하는 방법을 익힌다.

01 객체 포인터

- 객체 포인터 변수를 선언하는 기본 형식은 다음과 같다.

```
클래스명 *객체 포인터 변수;
```

객체 포인터 변수 기본 형식

- 객체 포인터 변수는 특정 객체 변수의 주소값을 저장한다.

```
Complex x(10, 20);  
Complex *pCom;  
pCom = &x;
```

- . 연산자는 객체로 멤버에 접근할 때 사용하고, -> 연산자는 객체 포인터로 멤버를 참조할 때 사용한다.

```
pCom->ShowComplex();
```

예제 11-1. 객체 포인터 사용하기(11_01.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real;
    int image;
public:
    Complex(int r = 0, int i = 0);
    void ShowComplex() const;
};
Complex::Complex(int r, int i) : real(r), image(i)
{
}
void Complex::ShowComplex() const
{
    cout << "( " << real << " + " << image << "i )"
        << endl;
}
```

```
void main() {
    Complex x(10, 20);
    Complex y;

    cout << " Object x => ";
    x.ShowComplex();
    cout << " Object y => ";
    y.ShowComplex();

    Complex *pCom;    // 객체 포인터 선언
    pCom = &x;         // 객체 x의 주소 할당

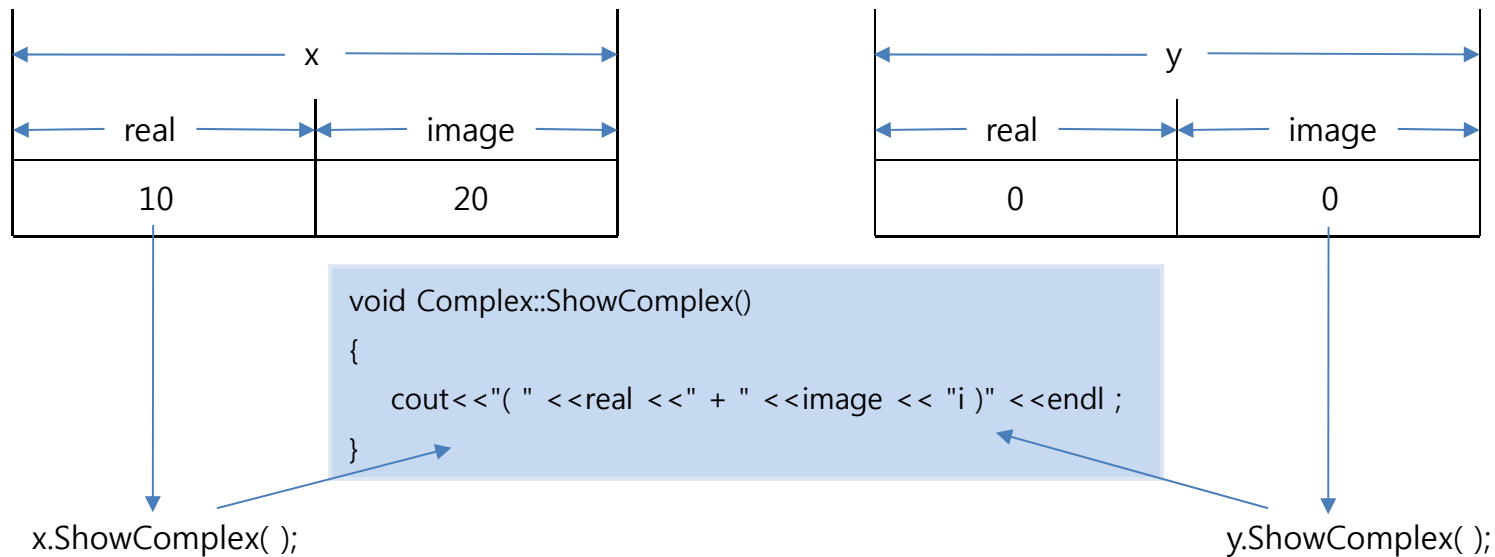
    cout << "\n pCom->ShowComplex() => ";
    pCom->ShowComplex();    // 객체 x의 멤버함수 호출

    pCom = &y;          // 객체 y의 주소 할당
    cout << " pCom->ShowComplex() => ";
    pCom->ShowComplex();    // 객체 y의 멤버함수 호출
}
```

01 객체 포인터

■ 객체 내의 멤버변수와 멤버함수의 구조

- 멤버함수를 호출하면 함수 내의 멤버변수는 특정 객체의 멤버변수가 된다.
멤버변수는 객체를 선언할 때마다 기억공간을 따로 할당 받아 관리된다.
- x.ShowComplex() 호출의 경우 ShowComplex 함수 내의 멤버가 객체 x의 멤버가 되고,
y.ShowComplex() 호출의 경우 ShowComplex 함수 내의 멤버가 객체 y의 멤버가 된다.

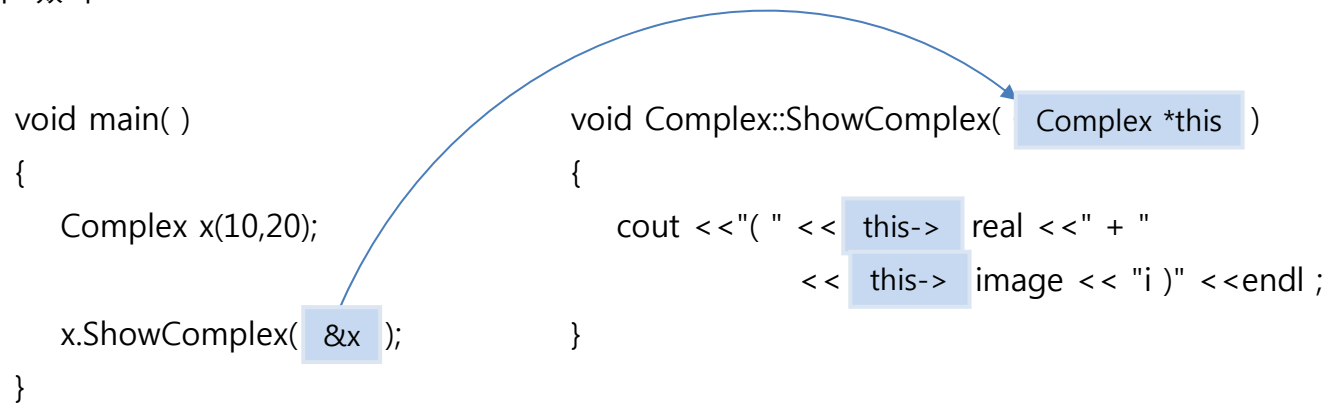


[그림 11-3] 멤버함수의 동작 원리

01 객체 포인터

■ 내부 포인터 this

- **this**는 컴파일러에 의해 생성되는 포인터로, 멤버함수를 호출한 객체를 가리키고 멤버함수 내에서만 사용할 수 있다.



[그림 11-4] 멤버함수에서의 멤버변수가 참조하는 객체의 정체

- 멤버변수는 멤버(구성원)이므로 '객체명.멤버' 혹은 '객체포인터->멤버'와 같이 누구의 멤버인지를 명시해야 하지만, 지금까지는 별 의심없이 멤버변수를 멤버함수 내에서 사용해왔다. 이는 컴파일러가 묵시적으로 모든 멤버 앞에 **this->**를 붙여주었기 때문이다.

01 객체 포인터

- **this 포인터를 정리하면 다음과 같다.**
 - ① this 포인터는 멤버함수 내에서 호출 객체의 주소를 저장하는 포인터다.
 - ② this 포인터는 컴파일러에 의해서 제공되므로 프로그래머가 별도로 선언하지 않아도 멤버함수 내에 항상 존재한다.
 - ③ 객체에 의해 멤버함수가 호출되면 컴파일러는 호출한 객체의 주소를 멤버함수 내의 this 포인터에 저장한다.
 - ④ 멤버함수 내에서 멤버변수를 참조하거나 다른 멤버함수를 호출할 때 묵시적으로 this 포인터로 접근한다.
 - ⑤ 컴파일러가 멤버변수 앞에 자동적으로 포인터를 붙여주므로 this 포인터를 생략하는 경우가 많다.
- this를 반드시 사용해야 하는 경우는 함수의 매개변수와 멤버 변수명이 동일해서 이를 구분해야 할 때다.

예제 11-2. this 포인터를 명시적으로 사용하기(11_02.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real;
    int image;
public:
    Complex(int real = 0, int image = 0);
    void ShowComplex() const;
};

/*
// 함수의 매개변수와 멤버변수의 이름이 같은 경우
Complex::Complex(int real, int image)
{
    real=real;
    image=image;
}
*/
```

```
// this 포인터를 이용해 함수의 매개변수와 멤버변수를 구분
Complex::Complex(int real, int image)
{
    this->real = real;
    this->image = image;
}

// 호출한 객체의 주소를 가지는 포인터 this
void Complex::ShowComplex() const
{
    cout << "( " << this->real << " + " << this->image << "i )"
        << endl;
}

void main()
{
    Complex x(10, 20);
    x.ShowComplex();
}
```

02 객체의 매개변수 전달 방식

① 객체의 값에 의한 전달 방식

- '값에 의한 전달 방식'은 함수를 호출할 때 기술한 실 매개변수의 값만 함수 측의 형식 매개변수로 전달된다.
즉, 형식 매개변수는 실 매개변수와는 별개의 기억공간이 할당되고 여기에 값만 복사된다.
- 객체의 반환
 - return문을 사용하여 객체를 반환할 수 있다.

② 객체의 주소에 의한 전달 방식

- 객체의 주소값을 함수에 전달하여 함수 내부에서 포인터로 간접 참조할 수 있다.

- ※ 동일한 클래스형으로 선언된 객체끼리는 대입 연산자로 멤버변수의 값을 치환할 수 있다.
즉, 구조체의 경우와 유사하게 객체 단위로 치환하면 객체 내의 모든 멤버변수의 값이 복사된다.

예제 11-4. 객체의 값에 의한 전달 방식의 함수 작성하기(11_04.cpp)_1

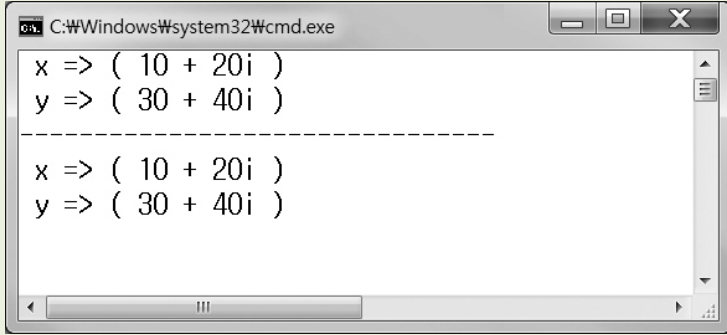
```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;
    void SetComplex(int r=0, int i=0);
};

Complex::Complex(int r, int i) {
    real=r;
    image=i;
}

void Complex::ShowComplex() const {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
```

예제 11-4. 객체의 값에 의한 전달 방식의 함수 작성하기(11_04.cpp)_2

```
void Complex::SetComplex(int r, int i) {  
    real=r;  
    image=i;  
}  
  
void CopyComplex(Complex des, Complex src) {    // 객체의 값에 의한 전달방식을 사용하는 객체 복사 함수  
    des=src;  
}  
  
void main() {  
    Complex x(10, 20);  
    Complex y(30, 40);  
  
    cout<<" x => " ;  
    x.ShowComplex();  
    cout<<" y => " ;  
    y.ShowComplex();  
  
    cout<<"----- \n" ;  
    CopyComplex(y, x);  
    cout<<" x => " ;  
    x.ShowComplex();  
    cout<<" y => " ;  
    y.ShowComplex();  
}
```



```
C:\Windows\system32\cmd.exe  
x => ( 10 + 20i )  
y => ( 30 + 40i )  
-----  
x => ( 10 + 20i )  
y => ( 30 + 40i )
```

// 객체의 값에 의한 전달방식이므로 함수종료 후 y의 멤버변수 값들은 변화가 없음

예제 11-6. 객체의 주소에 의한 전달 방식의 함수 작성하기(11_06.cpp)_1

```
#include <iostream>
using namespace std;
class Complex
{
private :
    int real;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;
    void SetComplex(int r=0, int i=0);
};
Complex::Complex(int r, int i) {
    real=r;
    image=i;
}
void Complex::ShowComplex() const {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
void Complex::SetComplex(int r, int i) {
    real=r;
    image=i;
}
```

예제 11-6. 객체의 주소에 의한 전달 방식의 함수 작성하기(11_06.cpp)_2

```
void CopyComplex(Complex *pDes, Complex src) { // 객체의 주소에 의한 전달방식을 사용하는 객체 복사 함수
```

```
    *pDes=src;
```

```
}
```

```
void main() {
```

```
    Complex x(10, 20);
```

```
    Complex y(30, 40);
```

```
    cout<<" x => " ;
```

```
    x.ShowComplex();
```

```
    cout<<" y => " ;
```

```
    y.ShowComplex();
```

```
    cout<<"----- \n" ;
```

```
    CopyComplex(&y, x);
```

```
    // 객체 y를 주소에 의해 전달하여 참조함
```

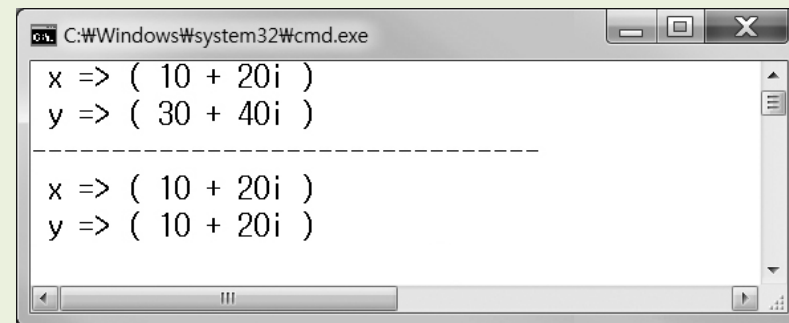
```
    cout<<" x => " ;
```

```
    x.ShowComplex();
```

```
    cout<<" y => " ;
```

```
    y.ShowComplex();
```

```
}
```



02 객체의 매개변수 전달 방식

③ 객체의 참조에 의한 전달 방식

- 참조 변수를 이용하면 실 매개변수의 값을 변경할 수 있다.
- CopyComplex 함수를 다음과 같이 변경해 보자.

```
void CopyComplex(Complex &des, const Complex &src)
{
    des=src;
}
```

예제 11-7. 객체의 참조에 의한 전달 방식의 함수 작성하기(11_07.cpp)_1

```
#include <iostream>
using namespace std;
class Complex
{
private :
    int real;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;
    void SetComplex(int r=0, int i=0);
};
Complex::Complex(int r, int i) {
    real=r;
    image=i;
}
void Complex::ShowComplex() const {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
void Complex::SetComplex(int r, int i) {
    real=r;
    image=i;
}
```


예제 11-7. 객체의 참조에 의한 전달 방식의 함수 작성하기(11_07.cpp)_2

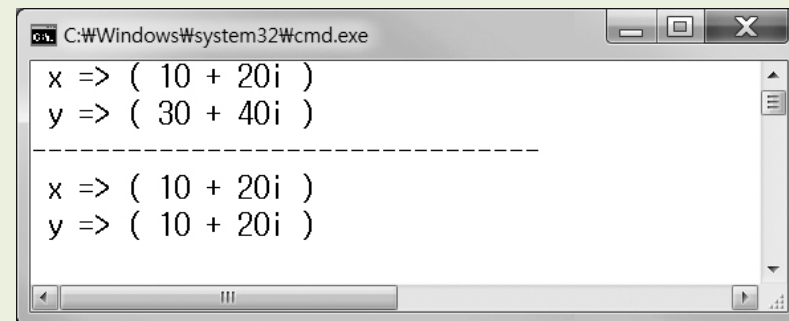
```
void CopyComplex(Complex &des, const Complex &src) { // 객체 참조에 의한 전달방식
    des = src;
}
```

```
void main() {
    Complex x(10, 20);
    Complex y(30, 40);

    cout << " x => ";
    x.ShowComplex();
    cout << " y => ";
    y.ShowComplex();

    CopyComplex(y, x);

    cout << " x => ";
    x.ShowComplex();
    cout << " y => ";
    y.ShowComplex();
}
```



```
C:\Windows\system32\cmd.exe
x => ( 10 + 20i )
y => ( 30 + 40i )
-----
x => ( 10 + 20i )
y => ( 10 + 20i )
```

03 정적 멤버변수와 정적 멤버함수

■ 정적 멤버변수

- 정적 멤버변수의 특징

- ① 정적 멤버변수는 클래스의 모든 인스턴스(객체)에 의해 공유된다.
- ② 정적 멤버변수의 원리는 전역변수와 동일하다. 하지만 정적 멤버변수는 해당 클래스명으로 접근해야 한다.

- 정적 멤버변수를 사용하기 위한 2가지 조건

- ① 정적 멤버변수는 특정 클래스 내부에 선언해야 한다.

객체를 선언하면 객체 단위로 멤버변수가 생성된다. 만약 하나의 클래스의 모든 객체 인스턴스들이 멤버변수 하나를 공유해야 한다면 ㉔처럼 **static** 예약어를 사용해서 정적 멤버변수를 생성한다.

- ② 정적 멤버변수는 클래스 밖에서 별도로 초기화되어야 한다.

정적 멤버변수는 클래스의 인스턴스와 상관없이 프로그램의 시작과 동시에 생성되는 변수로 ㉔처럼 클래스 밖에서 별도로 초기화해야 한다.

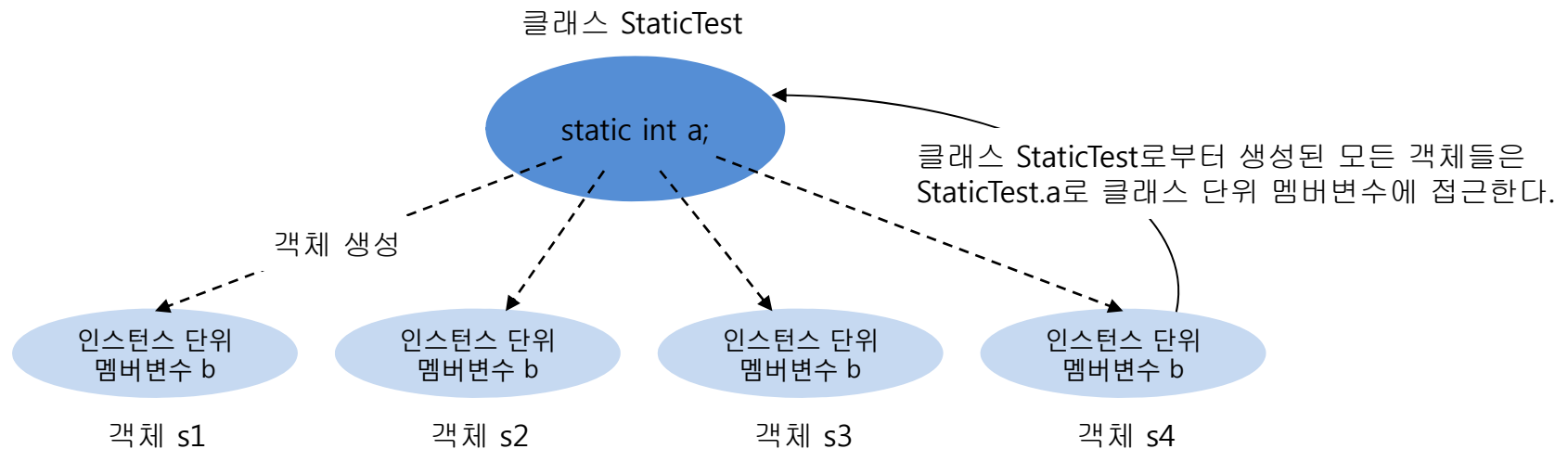
```
㉔ class StaticTest {  
    public:  
        static int a;  
        int b;  
}
```

```
㉔ int StaticTest::a=10;
```

03 정적 멤버변수와 정적 멤버함수

- StaticTest로 객체 4개를 생성한 예를 보자. 객체 4개를 생성하더라도 static으로 선언된 멤버들은 해당 클래스 당 한 개만 생성된다.

```
StaticTest s1, s2, s3, s4;
```



[그림 11-6] 정적 멤버변수, 클래스, 객체

- static 멤버변수는 여러 객체들에 의해서 공유되며 객체 생성 없이도 다음과 같이 클래스명으로 멤버에 접근할 수 있다.

```
StaticTest::a;
```

예제 11-8. 클래스 단위 멤버변수와 인스턴스 단위 멤버변수의 차이점 알아보기(11_08.cpp)

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

class StaticTest {
public:
    static int a; // 정적 멤버변수
    int b;        // 일반 멤버변수

    StaticTest(); // 생성자
};

StaticTest::StaticTest()
{
    b = 20;
}

int StaticTest::a = 10; // 정적 멤버변수 초기화
```

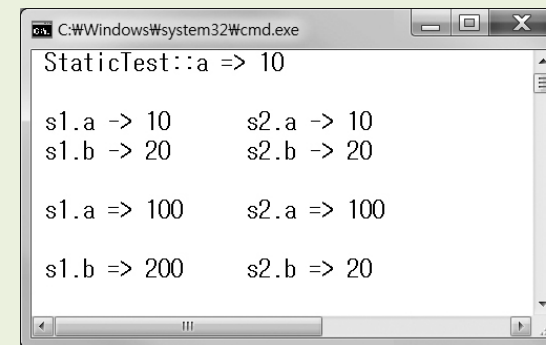
```
void main() {
    cout << " StaticTest::a : " << StaticTest::a << "\n\n";

    StaticTest s1, s2;

    cout << " s1.a : " << s1.a << "\t s2.a : " << s2.a << "\n";
    cout << " s1.b : " << s1.b << "\t s2.b : " << s2.b << "\n\n";

    s1.a = 100; // StaticTest 클래스에서 공유되는 정적 멤버변수의 값 변화
    cout << " s1.a : " << s1.a << "\t s2.a : " << s2.a << "\n";

    s1.b = 200; // 각 객체 인스턴스에 소속된 일반 멤버변수의 값 변화
    cout << " s1.b : " << s1.b << "\t s2.b : " << s2.b << "\n";
}
```



```
C:\Windows\system32\cmd.exe
StaticTest::a => 10

s1.a -> 10      s2.a -> 10
s1.b -> 20      s2.b -> 20

s1.a => 100     s2.a => 100
s1.b => 200     s2.b => 20
```

03 정적 멤버변수와 정적 멤버함수

■ 정적 멤버함수

- 정적 멤버변수를 **private**로 선언하면 이 정적 멤버변수를 사용하기 위한 멤버함수가 별도로 마련되어야 한다.

정적 멤버변수를 다루기 위한 멤버함수는 인스턴스 단위가 아닌 클래스 단위에서 사용 가능하도록 설계해야 하는데, 이를 위해 제공하는 것이 정적 멤버함수다.

```
static int GetA();
```

- 정적 멤버함수를 사용할 때의 주의사항
 - 정적 멤버함수의 특징
 - ❶ 정적 멤버함수에서는 **this** 포인터를 참조할 수 없다.
 - ❷ 정적 멤버함수에서는 인스턴스 멤버변수를 사용할 수 없다.
 - ❸ 정적 멤버함수는 오버라이딩되지 않는다 (추 후 설명).

예제 11-9. 정적 멤버함수 정의하기(11_09.cpp)

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

class StaticTest {
private:
    static int a;          // private 정적 멤버변수
    int b;
public:
    StaticTest();          // 생성자
    static void PrintA();   // 정적 멤버함수
    void PrintB();         // 일반 멤버함수
};

int StaticTest::a = 10;    // 정적 멤버변수 초기화

StaticTest::StaticTest()
{
    b = 20;
}
```

```
// 정적 멤버함수는 특정 인스턴스와 관계되지 않으므로,
// 즉, 해당 클래스의 모든 인스턴스와 공유되므로
// this 포인터를 사용할 수 없음.
// 또한, 특정 인스턴스의 멤버변수인 b는 클래스 전체적 단위로
// 호출되는 정적 멤버함수에서는 사용할 수 없다.
void StaticTest::PrintA() {
    cout << " a : " << a << endl;
    //cout <<" this->a : " << this->a << endl;
    //cout <<" b : " << b << endl;
}

void StaticTest::PrintB() {
    cout << " this->b : " << this->b << endl;
}

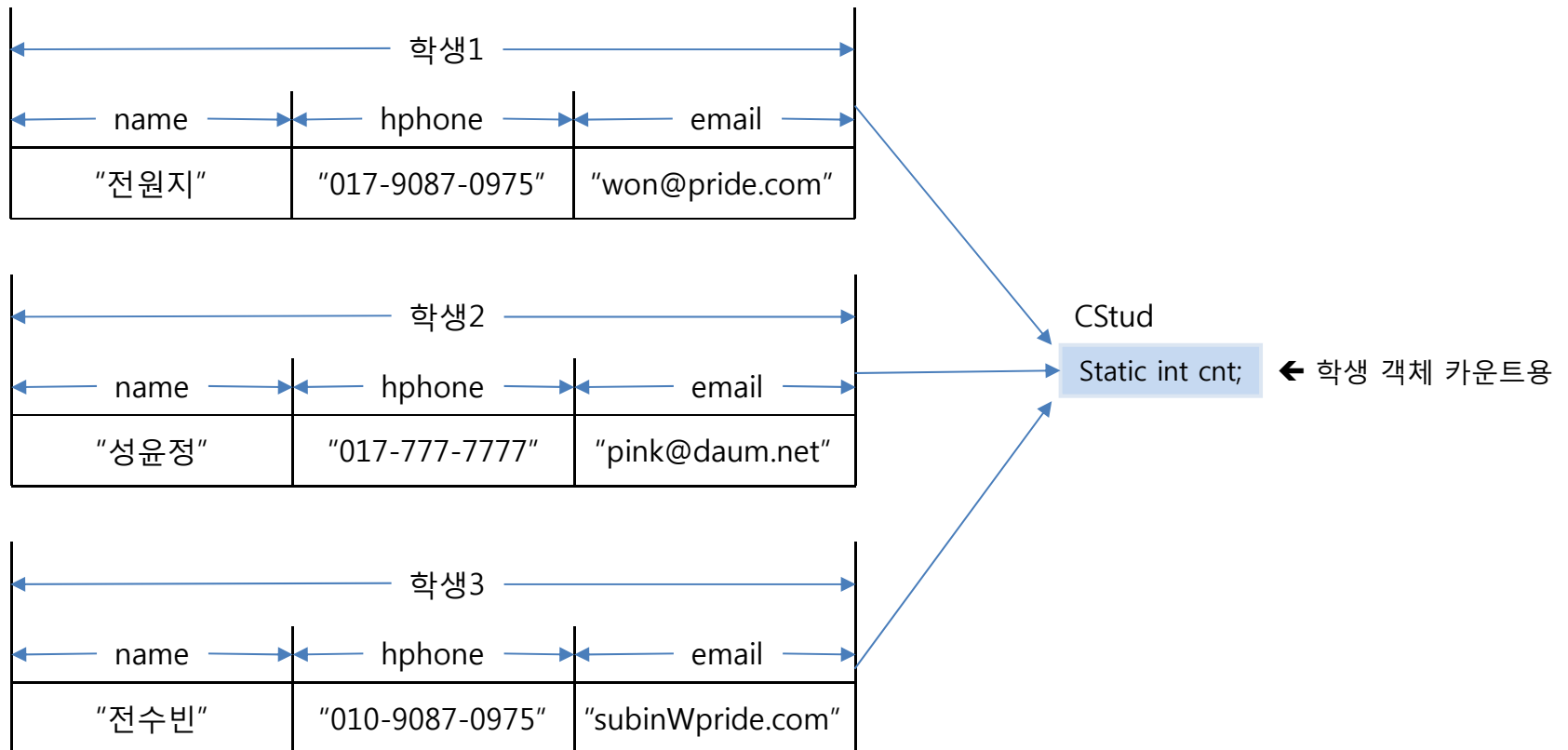
void main() {
    StaticTest s1;

    s1.PrintA();
    s1.PrintB();
}
```

03 정적 멤버변수와 정적 멤버함수

■ 정적 멤버변수의 유용한 사용 예

- 클래스를 설계하다 보면 인스턴스의 전체 개수를 세어야 하는 경우처럼 생성된 모든 인스턴스들이 멤버변수나 멤버함수를 공유해야 하는 경우가 있는데, 이럴 때는 static 예약어를 사용해야 한다.



[그림 11-7] 여러 객체에 의해서 공유되는 정적 멤버변수

예제 11-11. 인스턴스의 개수를 세는 정적 멤버변수 이용하기(11_11.cpp)_1

```
#include <iostream>
#include<string>    // 문자열 복사함수 사용을 위해
using namespace std;

class CStud {
private:
    char name[30];
    char hphone[20];
    char email[30];
    static int cnt;        // 정적 멤버변수
public:
    CStud(char *n = "성윤정", char *h = "017-777-7777", char *e = "pink@daum.net");
    ~CStud();
    void prn();
    static void prn_cnt(); // 정적 멤버함수
};

int CStud::cnt = 0;      // 정적 멤버변수 초기화
```


예제 11-11. 인스턴스의 개수를 세는 정적 멤버변수 이용하기(11_11.cpp)_2

```
CStud::CStud(char *n, char *h, char *e) {    // 생성자
    strcpy_s(name, n);    // 문자열 복사함수
    strcpy_s(hphone, h);
    strcpy_s(email, e);
    cnt++;                // 생성자 호출 시 객체 카운트 1개 증가
}

CStud::~CStud() {        // 소멸자
    cnt--;                // 소멸자 호출 시 객체 카운트 1개 감소
}

void CStud::prn() {
    cout << "이름 : " << name << endl;
    cout << "핸드폰 : " << hphone << endl;
    cout << "이메일 : " << email << endl << endl;
}

void CStud::prn_cnt() { // 정적 멤버변수를 사용하는 정적 멤버함수
    cout << "현재까지 등록된 인원수 : " << cnt << "WnWn";
}
```

예제 11-11. 인스턴스의 개수를 세는 정적 멤버변수 이용하기(11_11.cpp)_3

```
void main() {  
    // 객체가 선언되어 있지 않은 경우 클래스명으로 정적 멤버함수 호출  
    CStud::prn_cnt();  
  
    CStud man1("전수빈", "0110-9087-0975", "subin@pride.com");  
    man1.prn();  
    CStud man2("전원지", "017-9087-0975", "won@pride.com");  
    man2.prn();  
  
    cout << "Wn# 중간에 인원수를 파악합니다.";  
    man2.prn_cnt();    // 객체명으로도 정적 멤버함수를 호출할 수 있음  
  
    CStud man3;  
    man3.prn();  
  
    CStud::prn_cnt();    // 클래스명으로 정적 멤버함수 호출  
}
```

04 객체 배열

- 객체 배열을 선언할 때에는 클래스를 선언한 후 객체를 선언할 때 배열 형태로만 선언하면 된다.

```
클래스명 배열명[원소개수]; ;
```

객체 배열 선언 형식

- Complex 클래스로 원소 4개인 객체 배열을 생성해보자.

```
Complex arr[4];
```

- 객체 배열을 참조할 때는 객체 배열명 다음에 첨자를 적어서 원하는 위치의 원소를 지정하면 된다.

```
배열명[첨자].멤버변수;  
배열명[첨자].멤버함수;
```

객체 배열 참조 형식

- Complex 클래스에 매개변수가 2개인 생성자를 정의했다면 객체 배열의 원소를 초기화하려고 명시적으로 생성자를 호출할 수 있다.

```
Complex arr[4] = {  
    Complex(2, 4);  
    Complex(4, 8);  
    Complex(8, 16);  
    Complex(16, 32);  
};
```

예제 11-12. 객체 배열 사용하기(11_12.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;
};

Complex::Complex( int r, int i) : real(r), image(i)
{
}

void Complex::ShowComplex() const
{
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
```

```
void main() {
    Complex arr[4] = {
        Complex( 2, 4),
        Complex( 4, 8),
        Complex( 8, 16),
        Complex(16, 32),
    };

    for(int i = 0; i<4; i++)
        arr[i].ShowComplex();
}
```

04 객체 배열

① 객체 배열과 포인터

- 객체 포인터에 객체 배열명을 저장하면 객체 배열의 첫 번째 원소를 가리키게 된다.

```
Complex arr[4] = {  
    Complex( 2, 4),  
    Complex( 4, 8),  
    Complex( 8, 16),  
    Complex(16, 32),  
};  
  
Complex *pCom;  
pCom = arr; // pX = &arr[0];
```

- 객체 배열을 가리키는 객체 포인터를 그림으로 표현해보자.

pCom

1000번지



1000번지

arr[0]	2	4
arr[1]	4	8
arr[2]	8	16
arr[3]	16	32

04 객체 배열

- 객체 포인터 pCom은 배열의 시작 위치를 가리킨다. 그러므로 pCom으로 ShowComplex 멤버함수를 호출하면 배열의 첫 번째 원소인 arr[0]의 내용인 (2 + 4i)가 출력된다.

```
pCom->ShowComplex();
```

- 객체 포인터를 증가시켜서 원하는 위치의 객체 배열 원소의 주소값을 계산할 수 있다.

```
(pCom+1)->ShowComplex(); // arr[1]의 멤버함수가 호출되어 (4 + 8i)가 출력됨
```

② 함수의 매개변수로 객체 배열 사용하기

- 객체 배열의 시작 주소를 객체 포인터로 함수에 전달하여 포인터 연산을 통해 객체 배열의 원소에 접근한다.

예제 11-13. 객체 배열을 객체 포인터로 간접 참조하기(11_13.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;
};

Complex::Complex( int r, int i) : real(r), image(i)
{
}

void Complex::ShowComplex() const
{
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
```

```
void main() {
    Complex arr[4] = {
        Complex( 2, 4),
        Complex( 4, 8),
        Complex( 8, 16),
        Complex(16, 32),
    };

    Complex *pCom = arr;

    pCom->ShowComplex();
    (pCom+1)->ShowComplex();
}
```

05 프렌드 함수

- 일반 함수에서는 특정 class에 소속된 private 멤버를 직접 사용할 수 없다. 하지만, 이를 허용해 주는 함수가 프렌드 (friend) 함수다. 프렌드 함수는 데이터 은닉에 위배되므로 예외적인 경우에만 사용한다.
- 일반 함수를 특정 클래스의 프렌드 함수로 만들려면 다음과 같이 선언하면 된다.

friend 함수명(매개변수 리스트);

프렌드 함수 선언 형식

- 프렌드 함수가 되기 위한 조건은 다음과 같다.
 - ❶ 접근하고자 하는 private 멤버를 갖는 클래스 내부에 프렌드 함수를 선언한다.
 - ❷ 프렌드 함수를 선언할 때는 함수명 앞에 예약어 friend를 붙인다.

예제 11-15. 프렌드 함수 정의하기(11_15.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real ;
    int image;
public :
    Complex( int r=0, int i=0);
    void ShowComplex() const;

    friend void prn(Complex *pCom); // friend 함수
};

Complex::Complex( int r, int i) : real(r), image(i)
{ }

void Complex::ShowComplex() const {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}
```

```
void prn(Complex *pCom ) { // 일반함수
    for(int i=0; i<4; i++)
        cout<<"( " <<pCom[i].real <<" + "
            <<pCom[i].image<<"i )" <<endl ;
        // friend 일반함수에서 private 멤버를 직접 접근
    }

void main() {
    Complex arr[4] = {
        Complex( 2, 4),
        Complex( 4, 8),
        Complex( 8, 16),
        Complex(16, 32),
    };

    prn(arr); // friend 함수 호출 (일반함수로 호출)
}
```

06 객체를 다루기 위한 함수

① 두 객체의 합을 구하는 함수 구현하기

- Complex 객체(10, 20)에 Complex 객체(20, 40)을 더한 결과값이 Complex 객체(30, 60)이 되도록 정의하자.

두 Complex 객체의 합을 구하는 멤버함수 Sum을 정의하여 다음과 같이 호출해 본다.

```
Complex x(10, 20), y(20, 40), z;  
z=x.Sum(y);
```

- Sum 함수는 Complex 객체 x에 매개변수인 Complex 객체 y를 더해서 그 결과를 반환한다. 그리고 그 반환값을 Complex 객체 z에 저장한다. Complex 클래스의 멤버함수이므로 함수명 Sum 앞에는 Complex::를 붙인다.

```
Complex Complex::Sum(Complex rightHand);
```

- 객체명 앞에 붙은 x에 대한 정보는 포인터 this가 가지고 있고, 실 매개변수 y에 대한 정보는 형식 매개변수 rightHand에 전달된다. 함수 내부에서는 **this가 가리키는 객체(x)**와 **형식 매개변수 rightHand 객체(y)**의 멤버변수 real 끼리의 합을 구하고, 멤버변수 image 끼리의 합을 구해 결과값을 res에 저장한다.

```
Complex Complex::Sum(Complex rightHand)  
{  
    Complex res;  
    res.real = this->real + rightHand.real;  
    res.image = this->image + rightHand.image;  
    return res;  
}
```

예제 11-16. 두 복소수를 더하는 멤버함수 작성하기(11_16.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex(int r=0, int i=0);
    void ShowComplex();
    Complex Sum(Complex rightHand); // 객체 덧셈 함수
};

void Complex::ShowComplex() {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}

Complex::Complex( int r, int i) {
    real=r;
    image=i;
}
```

```
Complex Complex::Sum(Complex rightHand) {
    Complex res;
    res.real = this->real + rightHand.real; // x.real + y.real
    res.image = this->image + rightHand.image;
    return res;
}

void main() {
    Complex x(10,20), y(20, 40);
    Complex z;

    z = x.Sum( y );

    x.ShowComplex();
    y.ShowComplex();
    z.ShowComplex();
}
```

06 객체를 다루기 위한 함수

- 일반 함수를 사용해서 두 객체의 합을 구해보자.
- 일반 함수는 private 멤버 변수를 호출할 수 없기 때문에 프렌드 함수를 사용한다.
- 일반 함수에는 멤버함수에만 있는 포인터 this가 없기 때문에 피연산자 2개를 모두 매개변수로 받아야 한다.
복소수끼리 더한 결과는 복소수이므로 Complex 클래스형 이어야 한다.

```
Complex Sum(Complex leftHand, Complex rightHand)
{
    Complex res;
    res.real = leftHand.real + rightHand.real;
    res.image = leftHand.image + rightHand.image;
    return res;
}
```

예제 11-17. 두 복소수를 더하는 프렌드 함수 작성하기(11_17.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex(int r=0, int i=0);
    void ShowComplex();
    friend Complex Sum(Complex leftHand, Complex
rightHand); // friend 함수 선언
};
void Complex::ShowComplex() {
    cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
}
Complex::Complex( int r, int i) {
    real=r;
    image=i;
}
```

```
// friend로 지정된 일반함수를 이용한 객체 덧셈
Complex Sum(Complex leftHand, Complex rightHand) {
    Complex res;
    res.real = leftHand.real + rightHand.real;
    res.image = leftHand.image + rightHand.image;
    return res;
}

void main() {
    Complex x(10,20), y(20, 40);
    Complex z;

    // 두 객체를 매개변수로 가지는 일반함수 호출
    z = Sum( x, y );

    x.ShowComplex();
    y.ShowComplex();
    z.ShowComplex();
}
```

06 객체를 다루기 위한 함수

② 자신의 값을 1만큼 증가시키는 함수 구현하기

- **선행처리 방식**으로 피연산자를 1만큼 증가하는 AddOnePrefix를 멤버함수로 정의해보자.

```
Complex x(10,20), y(20, 40);  
Complex z;  
z=x.AddOnePrefix();
```

- ++ 연산자를 AddOnePrefix 함수로 **멤버함수로 구현**하면 포인터 **this**가 있으므로 매개변수를 지정하지 않아도 된다.

```
Complex Complex::AddOnePrefix()  
{  
    ++this->real;  
    ++this->image;  
    return *this;  
}
```

- AddOnePrefix 함수 내에서 피연산자로 사용된 x의 정보는 유일하게 this인데, 이 this는 포인터이므로 결과값으로 그대로 반환하면 안 되고 this가 가리키는 주소에 저장된 객체 값 전체를 넘겨주어야 한다.

```
return *this;
```

06 객체를 다루기 위한 함수

- **후행처리** 하는 ++ 연산자와 동일한 동작을 하는 AddOnePostfix 함수를 정의해 보자.

```
z=y.AddOnePostfix();
```

- 후행처리 ++ 연산자와 같이, 현재 객체를 우선 temp에 저장해 놓은 후
현재 객체의 멤버변수 값을 증가시키고, 최종적으로 값이 증가되기 전의 객체를 반환한다.

```
// c=b++; ← 후행처리의 예

Complex Complex::AddOnePostfix()
{
    Complex temp;
    temp=*this;
    ++this->real;
    ++this->image;
    return temp;    // 멤버변수의 값은 증가시키되, 반환되는
                   // 객체는 증가되기 전의 값을 갖는 객체를 반환
}
```

예제 11-18. 자신의 값을 1만큼 증가시키는 멤버함수 작성하기(11_18.cpp)_1

```
#include <iostream>
using namespace std;
class Complex {
private :
    int real;
    int image;
public :
    Complex(int r=0, int i=0);
    void ShowComplex();

    Complex AddOnePrefix();    // 선행처리
    Complex AddOnePostfix();  // 후행처리
};

Complex::Complex( int r, int i) {
    real=r;
    image=i;
}
```

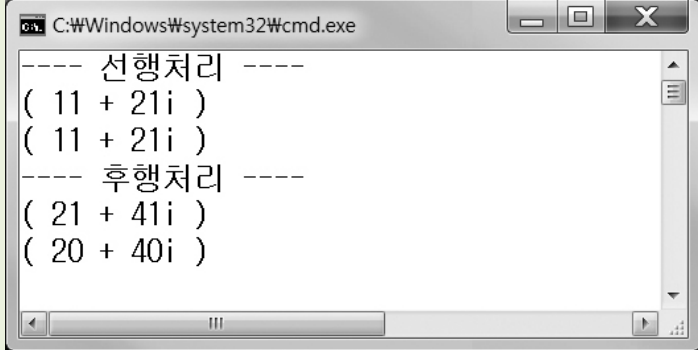
```
void Complex::ShowComplex() {
    cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
}

Complex Complex::AddOnePrefix() {
    ++this->real;
    ++this->image;
    return *this;
}

Complex Complex::AddOnePostfix() {
    Complex temp;
    temp=*this;
    ++this->real;
    ++this->image;
    return temp;
}
```


예제 11-18. 자신의 값을 1만큼 증가시키는 멤버함수 작성하기(11_18.cpp)_2

```
void main() {  
    Complex x(10,20), y(20, 40);  
    Complex z;  
  
    cout<<"---- 선행처리----\n";  
    z=x.AddOnePrefix(); // 증가된 후의 값을 z에 저장  
    x.ShowComplex();  
    z.ShowComplex();  
  
    cout<<"---- 후행처리----\n";  
    z=y.AddOnePostfix(); // 증가되기 전의 값을 z에 저장  
    y.ShowComplex();  
    z.ShowComplex();  
}
```



```
C:\Windows\system32\cmd.exe  
---- 선행처리 ----  
( 11 + 21i )  
( 11 + 21i )  
---- 후행처리 ----  
( 21 + 41i )  
( 20 + 40i )
```

07 연산자 오버로딩

① 연산자 오버로딩의 의미

- 연산자 오버로딩에 대해 정리하면 다음과 같다.
 - ① 연산자 오버로딩은 C++에서 기본 자료형으로 사용하는 연산자를 재정의하는 것이다.
 - ② C++에서는 연산자조차 함수로 취급하기 때문에 함수를 정의하는 방법과 동일한 방법으로 연산자를 오버로딩할 수 있다.
 - ③ 연산자를 함수의 형태로 오버로딩하기 때문에 오버로딩된 연산자를 연산자 함수라고도 한다.
 - ④ 연산자를 정의할 때 연산에 참여하는 피연산자는 매개변수로 구현된다.
 - ⑤ 연산자를 정의할 때 매개변수의 자료형에 의해 그 연산자를 사용할 수 있는 자료형이 결정된다.
- 연산자 함수명은 기본 자료형에 의해 사용되던 연산자를 operator 예약어와 함께 연산 기호로 표현한다.

연산자 정의 기본 형식

```
반환값 operator 연산자(매개변수1, 매개변수2, ...)  
{  
    함수의 본체  
}
```

07 연산자 오버로딩

② Complex 객체를 피연산자로 하는 연산자 오버로딩

- 수치형 자료에 사용되는 + 연산자를 Complex 객체에 대해서 연산하도록 연산자 오버로딩을 해 보자.

```
Complex x(10,20), y(20, 40), z;  
z = x + y;
```

- 두 객체를 더하려고 기술한 "z = x + y;" 즉, 컴파일러에 의해서는 호출되는 형태는:

```
z = x.operator+(y);
```

- 복소수의 덧셈을 구하는 멤버함수 Sum을 호출하는 구조와 동일하다.

```
z = x.Sum(y);
```

- + 연산자 함수의 정의 역시 Sum 함수와 동일하다.

일반적인 멤버함수로 구현한 예	연산자 함수로 구현한 예
<pre>Complex Complex::Sum(Complex rightHand) { ... }</pre>	<pre>Complex Complex::operator+(Complex rightHand) { ... }</pre>

07 연산자 오버로딩

- Complex 클래스에 대해서도 2가지 용도로 정의할 수 있는 “- 연산자”를 오버로딩 해보자.

```
Complex Complex::operator-(const Complex &rightHand) const
{
    Complex res;
    res.real = this->real - rightHand.real;
    res.image = this->image - rightHand.image;
    return res;
}
```

- 뺄셈은 피연산자의 위치가 중요하므로 순서에 신경을 써서 구현해야 한다. 왼쪽 피연산자인 객체 x에서 오른쪽 피연산자인 객체 y를 빼야 하므로 위치에 유의해서 멤버함수의 내용을 작성해야 한다.

```
z = x - y ; // z = x.operator-(y);
```

07 연산자 오버로딩

- 뺄셈이 아닌 “음수 기호로서의 - 연산자”를 정의해 보자. 변수 앞에 - 연산자를 기술했다면 부호는 바뀌지만 변수는 바뀌지 않는다. 클래스 Complex도 객체의 부호를 바꿀 수 있도록 - 연산자를 오버로딩 해야한다.

```
Complex Complex::operator-() const
{
    Complex res;
    res.real = -real ;
    res.image = -image ;
    return res;
}
```

예제 11-20. Complex 클래스의 연산자 오버로딩하기(11_20.cpp)_1

```
#include <iostream>
using namespace std;
class Complex
{
private :
    int real;
    int image;
public :
    Complex(int r=0, int i=0);
    void ShowComplex();
    Complex operator+(Complex rightHand);
    Complex operator-(const Complex &rightHand) const;
    Complex operator-() const;
};
```

```
void Complex::ShowComplex()
{
    if(image>0)
        cout<<"(" <<real <<"+" <<image << "i)" <<endl ;
    else if(image<0)
        cout<<"(" <<real << image << "i)" <<endl ;
    else
        cout<<real<<endl ;
}

Complex::Complex( int r, int i)
{
    real=r;
    image=i;
}
```

예제 11-20. Complex 클래스의 연산자 오버로딩하기(11_20.cpp)_2

```
Complex Complex::operator+(Complex rightHand) {
    Complex res;
    res.real = this->real + rightHand.real;
    res.image = this->image + rightHand.image;
    return res;
}

Complex Complex::operator-(const Complex &rightHand)
const {
    Complex res;
    res.real = this->real - rightHand.real;
    res.image = this->image - rightHand.image;
    return res;
}

Complex Complex::operator-() const {
    Complex res;
    res.real = -real ;
    res.image = -image ;
    return res;
}
```

```
void main() {
    Complex x(10,20), y(20, 40), z;
    cout<<"-- 두Complex 객체에대한덧셈--\n";
    z = x + y;
    x.ShowComplex();
    y.ShowComplex();
    z.ShowComplex();

    cout<<"\n-- 두Complex 객체에대한뺄셈--\n";
    z = x - y;
    x.ShowComplex();
    y.ShowComplex();
    z.ShowComplex();

    cout<<"\n-- Complex 객체의부호변경--\n";
    z=-x;
    x.ShowComplex();
    z.ShowComplex();
}
```

07 연산자 오버로딩

④ 연산자를 오버로딩할 때의 주의사항

① C++에서 이미 사용하던 연산자만 오버로딩할 수 있다.

② 이항 연산자는 이항 연산자로, 단항 연산자는 단항 연산자로만 오버로딩할 수 있다. 예를 들면 `10%4`와 같이 `%` 연산자는 이항 연산자 형태로 오버로딩해야 한다. 다음은 잘못된 예다.

```
int a;
```

```
% a; // 나머지를 구하는 연산자를 단항 연산자로 사용하지 못한다.
```

③ C++에서 사용하는 연산자 중에서 다음 연산자는 오버로딩할 수 없다.

`.(멤버 참조 연산자), ::(스코프 연산자), ?:(조건 연산자), sizeof(sizeof 연산자), *(포인터 연산자)`

④ 연산자를 오버로딩하려면 피연산자가 적어도 하나 이상은 사용자 정의 자료형 이어야 한다.

다음은 잘못된 예다.

```
double operator+(double x, double y) // 잘못된 연산자 오버로딩
```

⑤ 대부분의 연산자는 멤버함수 또는 프렌드 함수로 오버로딩할 수 있다. 그러나 다음 연산자는 멤버함수로만 오버로딩 할 수 있다.

`=`(대입 연산자), `()`(함수 호출 연산자), `[]`(첨자 지정 연산자), `->`(객체 포인터에 대한 멤버 참조 연산자)

Homework

- Chapter 11 Exercise: 1, 2, 3, 4, 5, 6 ,7 ,8, 9