원리를 알면 IT가 맛있다
IT COOKBOOK

IT CookBook, C++ 하이킹 객체지향과 만나는 여행

[강의교안 이용 안내]

C++ 하이킹

객체지향과 만나는 여행

Chapter 11. 객체의 다양한 활용

한빛아카데미 Hanbit Academy, Inc.

목차

학습목표

- 클래스로 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.

객체 포인터 변수 기본 형식

```
클래스명 *객체 포인터 변수;
```

- 객체 포인터 변수는 특정 객체 변수의 주소값을 저장하고 있어야 한다.

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

- 구조체처럼 클래스에서도 멤버참조 연산자가 2개 존재한다. 바로 . 연산자와 -> 연산자다. . 연산자는 객체로 멤버에 접근할 때 사용하고, -> 연산자는 객체 포인터로 멤버를 참조할 때 사용한다.

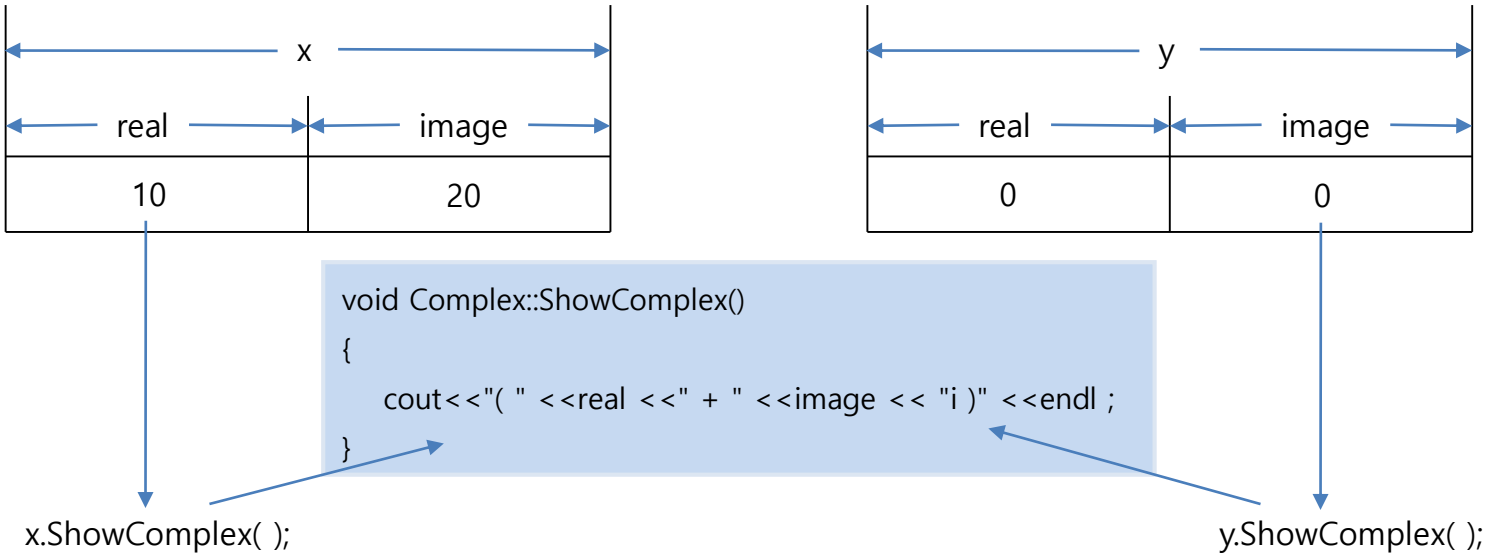
```
pCom->ShowComplex();
```

객체 내의 멤버변수와 멤버함수의 구조

- 멤버함수를 호출하면 함수 내의 멤버변수는 특정 객체의 멤버변수가 된다. 멤버변수는 객체를 선언할 때마다 기억공간을 따로 할당받아 관리된다. 멤버함수는 멤버변수처럼 따로 관리되지 않고 호출할 때 어떤 객체로 접근했느냐에 따라 함수 내에서 출력될 멤버변수의 소속 객체가 정해진다.

01 객체 포인터

- ShowComplex 함수를 호출할 때 객체 x로 접근했다면 ShowComplex 함수 내의 멤버가 객체 x의 멤버가 되고, ShowComplex 함수가 객체 y로 접근해서 호출되면 그 때는 ShowComplex 함수 내의 멤버가 객체 y의 멤버가 된다.

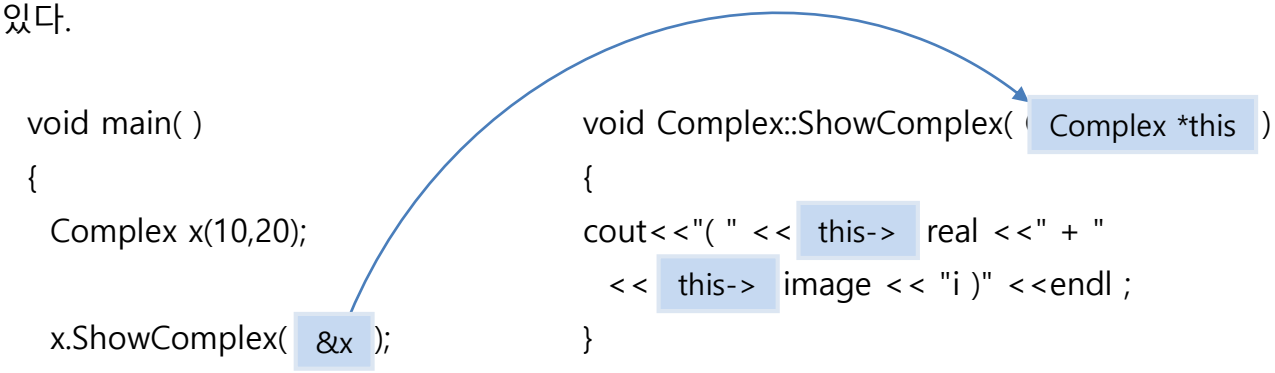


[그림 11-3] 멤버함수의 동작 원리

01 객체 포인터

내부 포인터 this

- this는 컴파일러에 의해 생성되는 포인터로, 멤버함수를 호출한 객체를 가리키고 멤버함수에서만 사용할 수 있다.

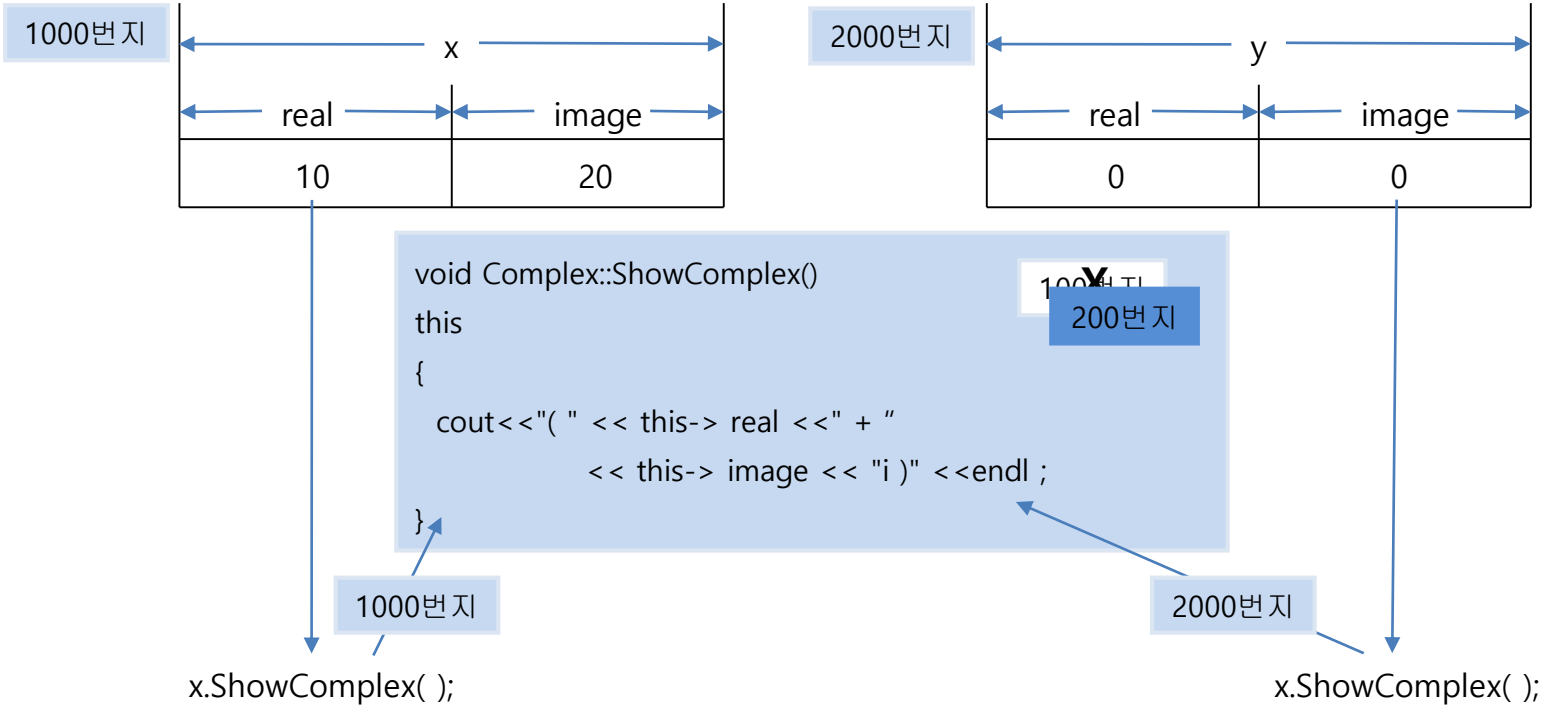


[그림 11-4] 멤버함수에서의 멤버변수가 참조하는 객체의 정체

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

01 객체 포인터

- this 포인터를 직접 명시해서 코드 블록에 한 개만 존재하는 멤버함수를 공유하는 원리를 알아보자.



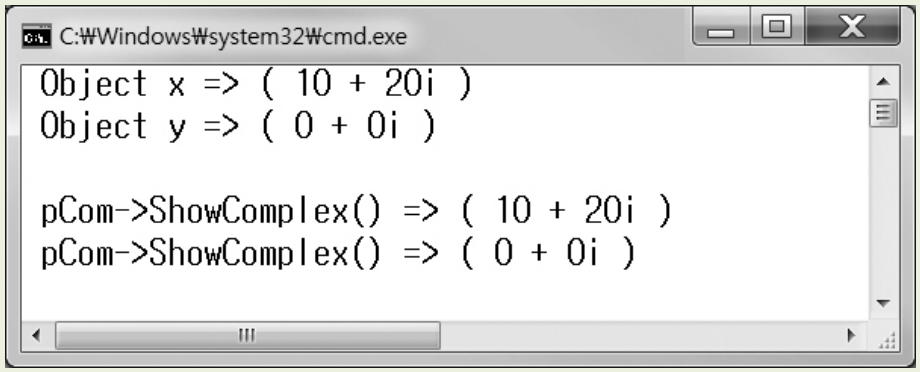
[그림 11-5] 멤버함수가 여러 객체에 의해서 공유되는 원리

01 객체 포인터

- this 포인터를 정리하면 다음과 같다.
  ❶ this 포인터는 멤버함수 내에서 호출 객체의 주소를 저장하는 포인터다.
  ❷ this 포인터는 컴파일러에 의해서 제공되므로 프로그래머가 별도로 선언하지 않아도 멤버함수 내에 항상 존재한다.
  ❸ 객체에 의해 멤버함수가 호출되면 컴파일러는 호출한 객체의 주소를 멤버함수 내의 this 포인터에 저장한다.
  ❹ 멤버함수 내에서 멤버변수를 참조하거나 다른 멤버함수를 호출할 때 묵시적으로 this 포인터로 접근한다.
  ❺ 컴파일러가 멤버변수 앞에 자동적으로 포인터를 붙여주므로 this 포인터를 생략하는 경우가 많지만 프로그래머가 this 포인터를 멤버변수나 멤버함수에 직접 붙여주어도 상관없다.
- 객체 포인터 this를 반드시 사용해야 하는 경우는 함수의 매개변수와 멤버변수명이 동일해서 이를 구분해야 할 때다.

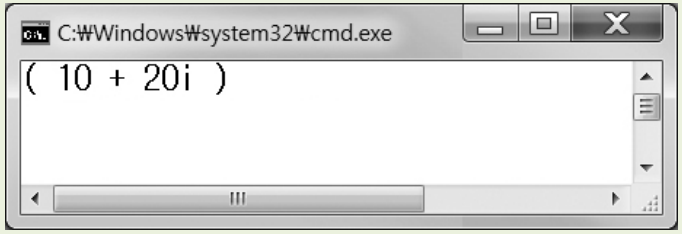
예제 11-1. 객체 포인터 사용하기(11_01.cpp)

책의 소스코드 참고



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

책의 소스코드 참고



02 객체의 매개변수 전달 방식

- 동일한 클래스형으로 선언된 객체끼리는 대입 연산자로 값을 치환할 수 있다. 객체 단위로 치환하면 객체 내의 모든 멤버변수의 값이 복사된다.

객체의 값에 의한 전달 방식

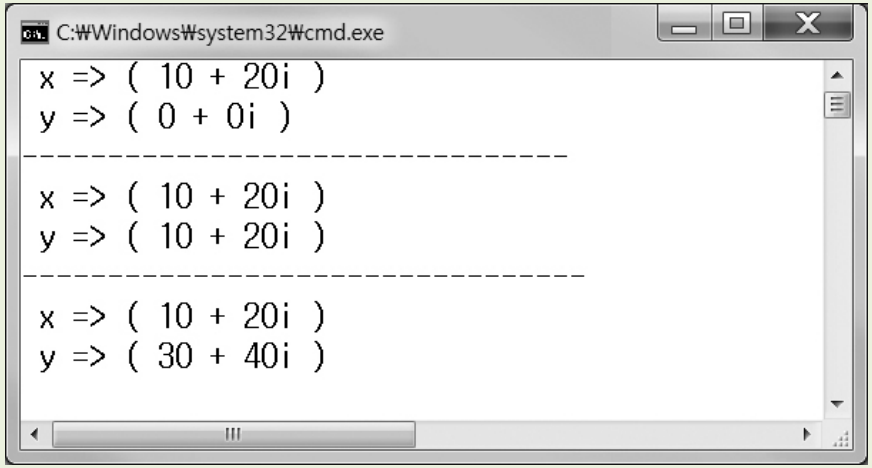
- '값에 의한 전달 방식'은 함수를 호출할 때 기술한 실 매개변수의 값만 함수 측의 형식 매개변수로 전달된다. 즉, 형식 매개변수는 실 매개변수와는 별개의 기억공간이 할당되고 여기에 값만 복사된다.
- 객체를 함수의 결과값으로 사용하기
  - 정수형이나 실수형 같은 기본 자료형을 함수의 결과값으로 반환할 수 있듯이 return문 다음에 객체를 기술해서 반환할 수 있다. 그런데 함수의 자료형은 return문 다음에 기술한 값과 같아야 하므로 함수의 자료형이 클래스로 선언된다.

객체의 주소에 의한 전달 방식

- 매개변수가 복사 대상이라면 변숫값이 함수 호출 후에 변경되어야 한다. 그리고 이렇게 함수 호출 후에 변경되어야 하는 실 매개변수는 객체의 주소값을 전달해주어 함수 내부에서 포인터로 간접 참조할 수 있도록 해야 한다.

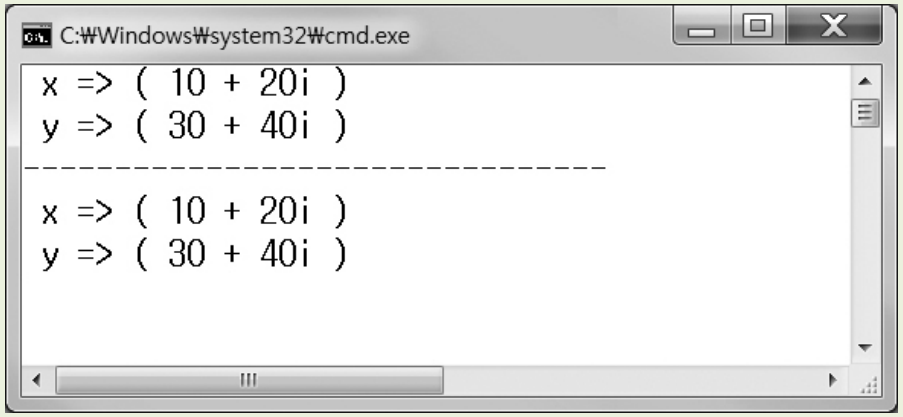
예제 11-3. 객체 단위로 값 치환하기(11_03.cpp)

책의 소스코드 참고



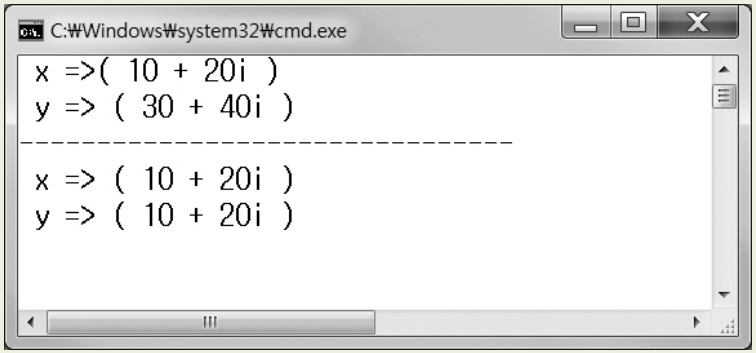
예제 11-4. 객체의 값에 의한 전달 방식의 함수 작성하기(11_04.cpp)

책의 소스코드 참고



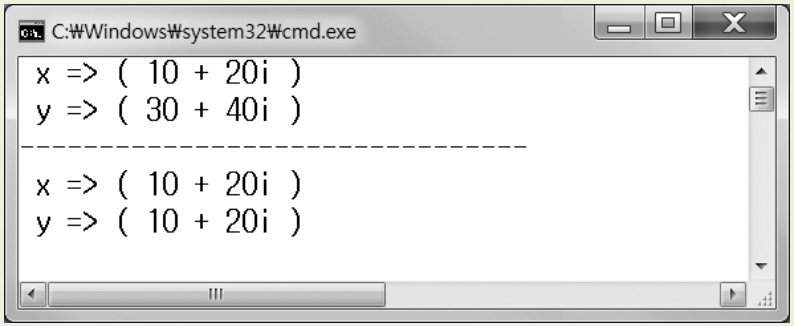
예제 11-5. 결과값이 객체인 함수 작성하기(11_05.cpp)


```
29 Complex CopyComplex(Complex des, Complex src)
30 {
31 des=src;
32 return des;
33 }
34 void main()
35 {
36 Complex x(10, 20);
37 Complex y(30, 40);
38
39 cout<<" x =>" ;
40 x.ShowComplex();
41 cout<<" y => " ;
42 y.ShowComplex();
43
44 cout<<"------------------------------ ₩n" ;
45 y=CopyComplex(y, x);
46 cout<<" x => " ;
47 x.ShowComplex();
48 cout<<" y => " ;
49 y.ShowComplex();
50 }
```



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


```
29 void CopyComplex(Complex *pDes, Complex src)
30 {
31 *pDes=src;
32 }
33 void main()
34 {
35 Complex x(10, 20);
36 Complex y(30, 40);
37
38 cout<<" x => " ;
39 x.ShowComplex();
40 cout<<" y => " ;
41 y.ShowComplex();
42
43 cout<<"------------------------------ ₩n" ;
44 CopyComplex(&y, x);
45 cout<<" x => " ;
46 x.ShowComplex();
47 cout<<" y => " ;
48 y.ShowComplex();
49 }
```



02 객체의 매개변수 전달 방식

객체의 참조에 의한 전달 방식

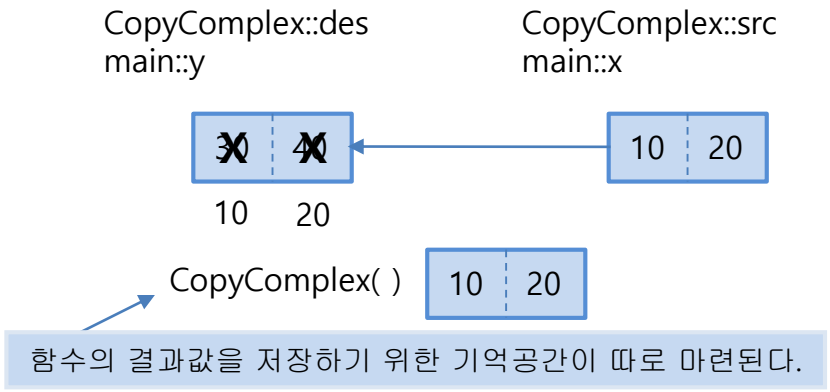
- 참조 변수를 이용하면 실 매개변수의 값을 변경할 수 있다는 장점이 있다. 또한 값에 의한 전달 방식은 메모리를 실 매개변수와 별개의 공간으로 만들기 때문에 함수가 호출될 때마다 메모리를 매번 할당하지만 참조 변수는 이름만 부여되기 때문에 메모리를 효율적으로 사용할 수 있다.
- CopyComplex 함수를 다음과 같이 변경해 보자.

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

- 함수의 결과값을 Complex 객체에 저장하려면 함수의 반환값도 Complex형이어야 한다. 함수도 값을 가질 수 있는데, 함수의 반환값으로 다음과 같이 선언하면 함수의 결과값을 저장하기 위한 기억공간이 따로 마련된다.

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

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



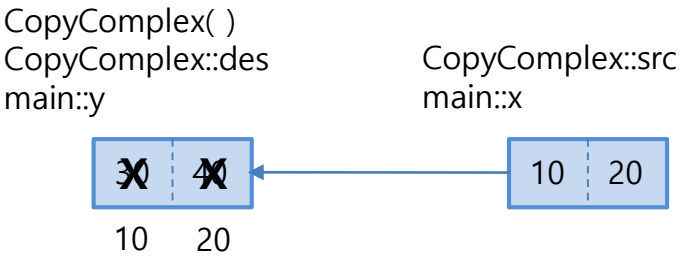
02 객체의 매개변수 전달 방식

- 메모리를 효율적으로 사용하려면 함수의 반환값도 참조 변수처럼 선언할 수 있다. 이렇게 선언하게 되면 함수의 결과값을 저장할 기억공간이 따로 마련되지 않고 return문 다음에 기술한 변수의 별칭이 된다.

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

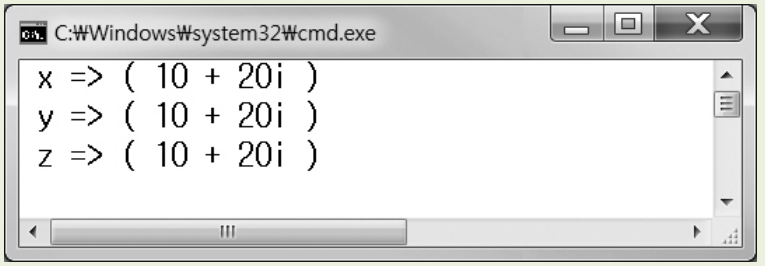
❷
```
void main()
{
  Complex x(10, 20);
  Complex y(30, 40);
  CopyComplex(y, x);
}
```

CopyComplex 함수는 des의 별칭이 되고 des는 x의 별칭이므로 하나의 기억공간이 3개의 이름으로 사용된다.



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


```
29 Complex & CopyComplex(Complex &des, const Complex &src)
30 {
31 des=src;
32 return des;
33 }
34 void main()
35 {
36 Complex x(10, 20);
37 Complex y(30, 40);
38 Complex z;
39
40 z=CopyComplex(y, x);
41 cout<<" x => " ;
42 x.ShowComplex();
43 cout<<" y => " ;
44 y.ShowComplex();
45 cout<<" z => " ;
46 z.ShowComplex();
47 }
```



03 정적 멤버변수와 정적 멤버함수

정적 멤버변수

- 정적 멤버변수의 특징
  - ❶ 정적 멤버변수는 클래스의 모든 인스턴스(객체)에 의해 공유된다.
  - ❷ 정적 멤버변수에 자료가 저장되어 값이 유지되는 원리는 전역변수와 동일하다. 하지만 정적 멤버변수는 해당 클래스명으로 접근해야 한다는 점에서 전역변수와 차이가 난다.
- 정적 멤버변수를 사용하기 위한 2가지 조건
  - ❶ 정적 멤버변수는 특정 클래스 내부에 선언해야 한다.
    클래스로 객체를 선언하면 객체 단위로 멤버변수가 생성된다. 만약 모든 객체 인스턴스들이 멤버변수 하나를 공유해야 한다면 ⓐ처럼 static 예약어를 사용해서 멤버변수를 생성한다.
  - ❷ 정적 멤버변수는 클래스 밖에서 별도로 초기화되어야 한다.
    정적 멤버변수는 클래스의 인스턴스와 상관없이 프로그램의 시작과 동시에 생성되는 변수로 ⓑ처럼 클래스 밖에서 별도로 초기화해야 한다.

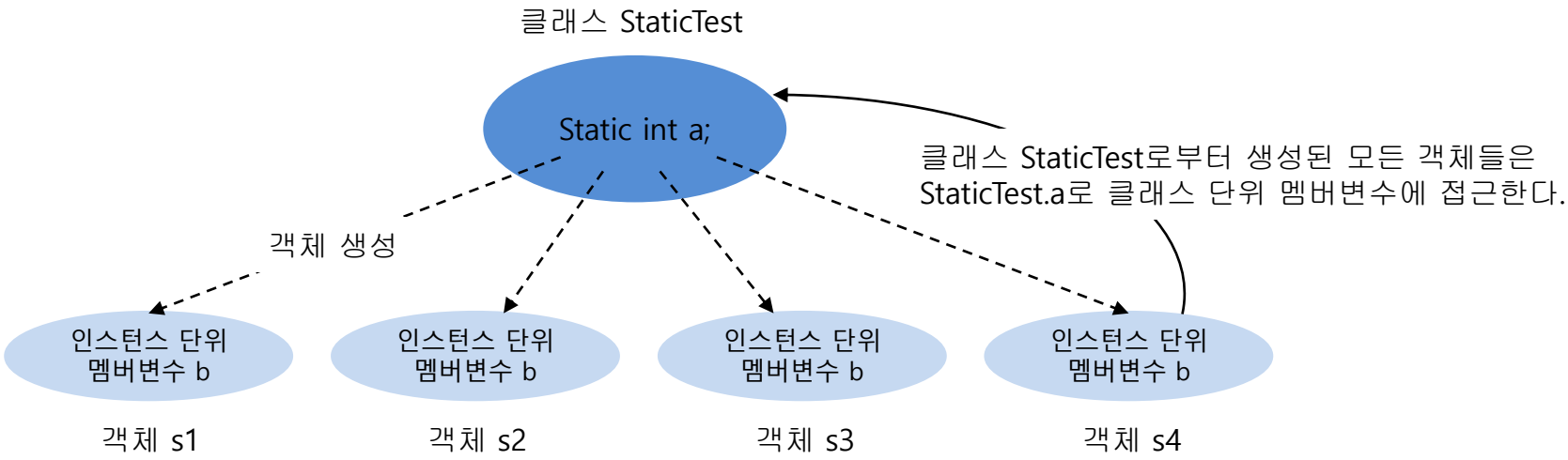
ⓐ
```
class StaticTest{
  public:
   static int a;
   int b;
}
```

ⓑ
```
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;
```



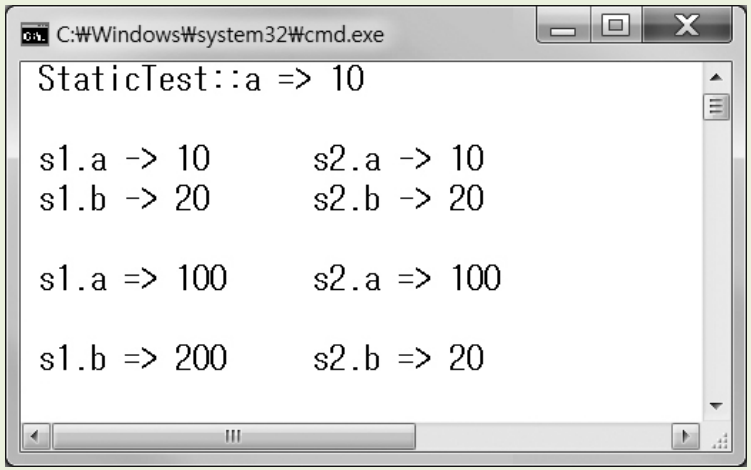
[그림 11-6] **정적 멤버변수, 클래스, 객체**

- static 멤버변수는 여러 객체들에 의해서 공유되며 객체 생성 없이도 다음과 같이 클래스명으로 멤버에 접근할 수 있다. 프로그램을 실행하면 객체가 생성되기 전 에 클래스가 메모리에 먼저 올라간다. 그런데 이때 static으로 선언된 멤버들도 메모리에 함께 올라가므로 클래스명으로 직접 접근할 수 있는 것이다.

```
StaticTest::a;
```

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

책의 소스코드 참고



03 정적 멤버변수와 정적 멤버함수

정적 멤버함수

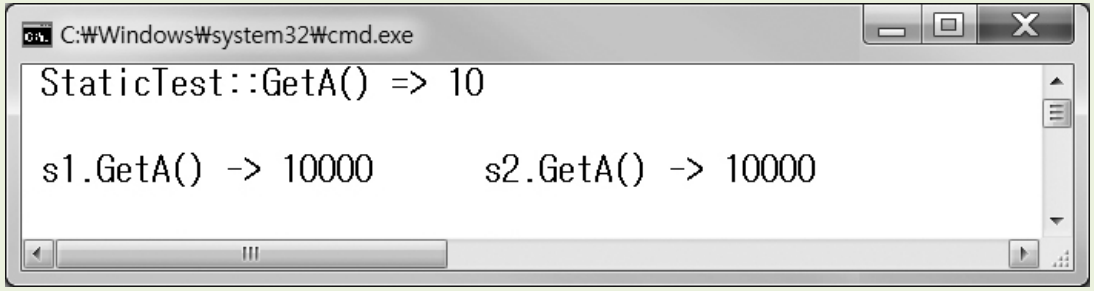
- 정적 멤버변수를 private로 선언하면 이 정적 멤버변수를 사용하기 위한 멤버함수가 별도로 마련되어야 한다. 정적 멤버변수를 다루기 위한 멤버함수는 인스턴스 차원이 아닌 클래스 차원에서 사용 가능하도록 설계해야 하는데, 이를 위해 제공하는 것이 정적 멤버함수다.

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

- 정적 멤버함수를 사용할 때의 주의사항
  - 정적 멤버함수의 특징
    1. 정적 멤버함수에서는 this 포인터를 참조할 수 없다.
    2. 정적 멤버함수에서는 인스턴스 변수를 사용할 수 없다.
    3. 정적 멤버함수는 오버라이딩되지 않는다.

예제 11-9. 정적 멤버함수 정의하기(11_09.cpp)

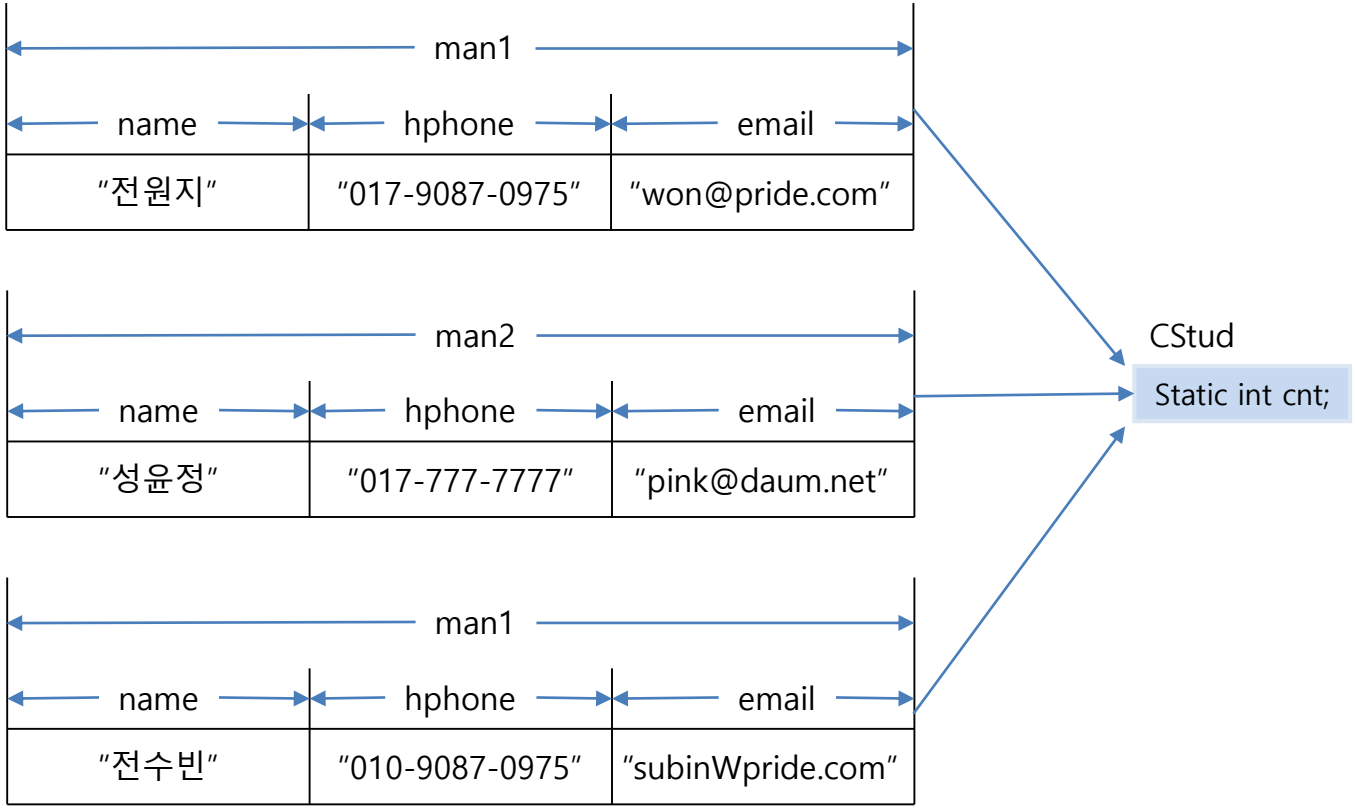
책의 소스코드 참고



03 정적 멤버변수와 정적 멤버함수

정적 멤버변수의 유용한 사용 예

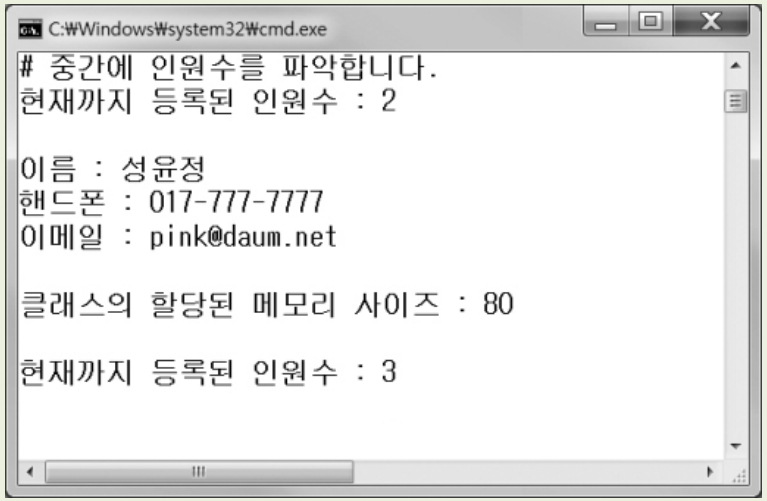
- 클래스를 설계하다 보면 인스턴스의 전체 개수를 세어야 하는 경우처럼 생성된 모든 인스턴스들이 멤버변수나 멤버함수를 공유해야 하는 경우가 있는데, 이럴 때는 static 예약어를 사용해야 한다.



[그림 11-7] 여러 객체에 의해서 공유되는 정적 멤버변수

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

책의 소스코드 참고



04 객체 배열

- 객체 배열을 선언할 때에는 클래스를 선언한 후 객체를 선언할 때 배열 형태로만 선언하면 된다. 그리고 객체 배열명 다음에 대괄호 안에 원소의 개수를 적어준다.

객체 배열 선언 형식

```
클래스명 객체_배열명[원소_개수];
```

- Complex 클래스로 원소 4개인 객체 배열을 생성해보자.

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

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

객체 배열 참조 형식

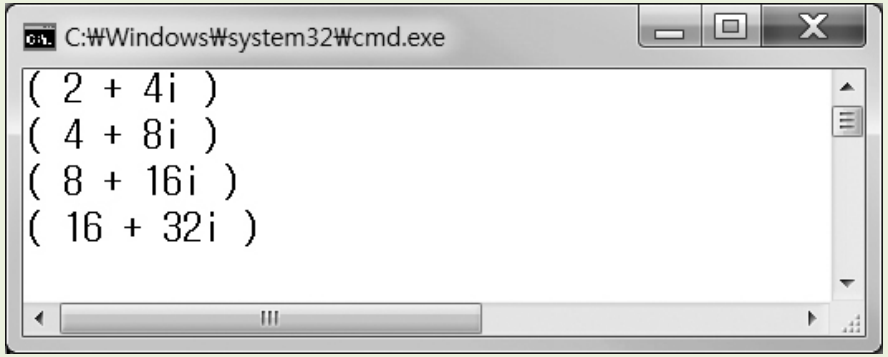
```
객체_배열명[첨자].멤버변수;
객체_배열명[첨자].멤버함수;
```

- 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)

책의 소스코드 참고



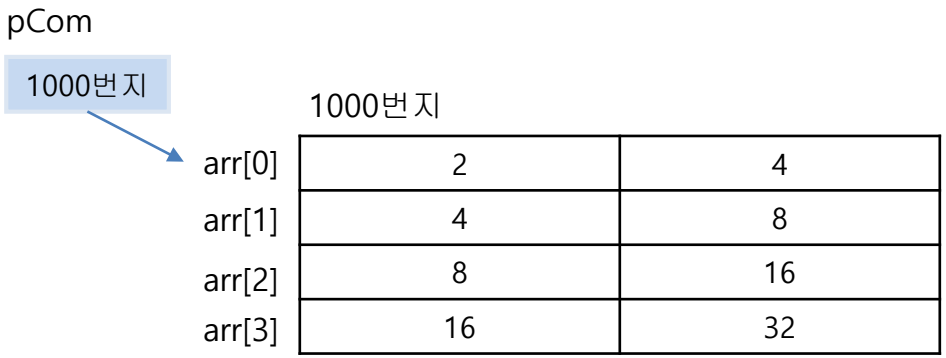
04 객체 배열

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.



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();
```

- pCom은 클래스 Complex로 선언된 객체 포인터이므로 객체가 차지하는 기억공간의 크기만큼 주소값이 증가하므로 객체 포인터를 증가시켜서 원하는 위치의 객체 배열 원소의 주소값을 계산할 수 있다.

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

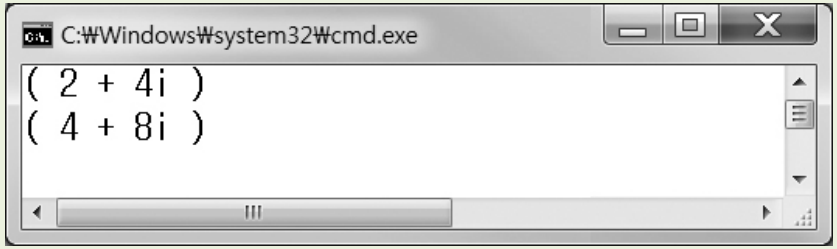
- 포인터 변수 pCom에 1을 더하면 다음 원소의 주소값이 계산된다. pCom+1로 다음 원소의 주소값을 계산한 후에 그 위치의 멤버변수를 참조하기 위한 -> 연산자가 연산되어야 하므로 pCom+1을 소괄호로 묶어준다. (pCom+1)->ShowComplex( )와 같이 포인터 변수에 1을 더한 후 멤버함수 ShowComplex를 호출하면 (4 + 8i)가 출력된다.

함수의 매개변수로 객체 배열 사용하기

- 객체 배열을 함수의 매개변수로 지정할 경우 객체 배열 전체의 값을 사용자 정의 함수로 넘겨줄 수 없다. 객체 배열의 시작 주소만 전달해 주면 이를 사용자 정의 함수에서 객체 포인터로 선언된 형식 매개변수가 받아 포인터 연산자인 +와 함께 주소를 증가시켜가면서 객체 배열의 모든 원소값을 출력한다.

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

책의 소스코드 참고



05 프렌드 함수

- 일반 함수에서는 private 멤버를 사용할 수 없지만 굳이 사용해야 한다면 이를 허용해 주는 함수가 프렌드(friend) 함수다. 클래스가 private 멤버를 두는 이유는 그 클래스 내의 멤버함수들만 접근할 수 있도록 하기 위해서다. 그렇지만 친구라며 접근하는 함수가 있으면 거절을 못하고 private 멤버를 사용하도록 허용하는 것이다. 프렌드 함수는 데이터 은닉에 위배되므로 정말 어쩔 수 없는 예외적인 경우에만 사용해야 한다.
- 일반 함수를 특정 클래스의 프렌드 함수로 만들려면 다음과 같이 선언하면 된다.

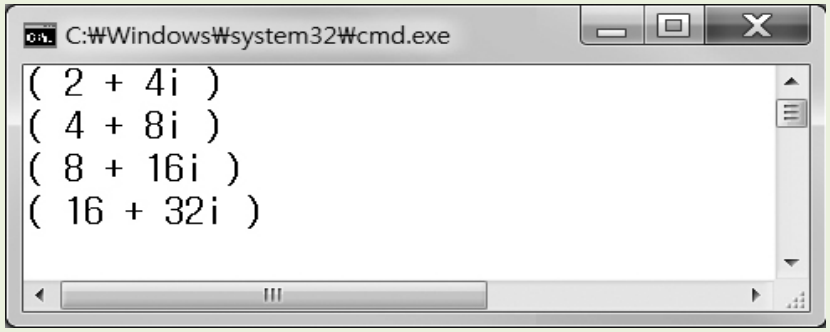
프렌드 함수 선언 형식

```
friend 프렌드 함수명(매개변수 리스트);
```

- 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)

책의 소스코드 참고



06 객체를 다루기 위한 함수

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 객체에 대한 합을 구하는 함수가 Sum이라는 이름의 멤버함수로 정의되어 있다면 다음과 같이 호출할 수 있다.

```
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 저장해야 한다. 함수의 결과값이 Complex 객체 z에 저장되어야 하므로 함수의 자료형은 Complex형이다. 멤버함수이므로 함수명 Sum 앞에는 Complex::를 붙여야 한다.

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

- 객체명 앞에 붙은 x에 대한 정보는 포인터 this가 가지고 있으므로 실 매개변수 y에 대한 정보는 형식 매개변수 rightHand에 전달된다.

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

06 객체를 다루기 위한 함수

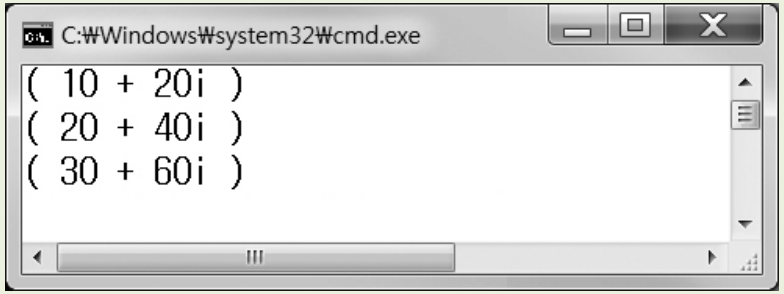
- Sum 함수 내에서는 두 객체에 대한 합을 저장할 Complex 객체가 함수 Sum 내부에 지역 변수로 선언(Complex res;)되어 있어야 한다. 두 Complex 객체의 합은 real, image 멤버변수끼리의 합을 의미한다. this는 + 연산자를 기준으로 왼쪽 피연산자의 정보를 받아오고, 매개변수는 오른쪽 피연산자의 정보를 받아오게 된다.
- 함수 내부에서는 this가 가리키는 객체와 형식 매개변수 rightHand 객체의 멤버변수 real은 real끼리의 합을 구하고, 멤버변수 image는 image끼리의 합을 구해 결과값을 res에 저장한다.

- 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 클래스형이어야 한다.

```
Complex Sum(Complex leftHand, Complex rightHand)
{
...
}
```

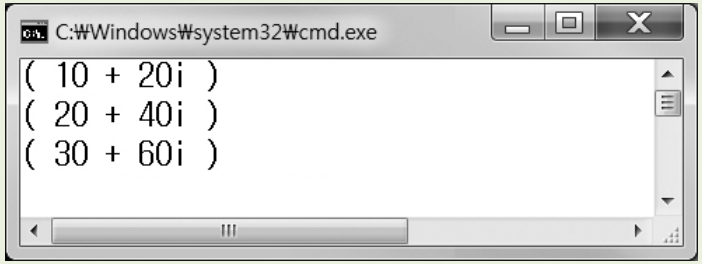
예제 11-16. 두 복소수를 더하는 멤버함수 작성하기(11_16.cpp)

책의 소스코드 참고



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

책의 소스코드 참고



06 객체를 다루기 위한 함수

자신의 값을 1만큼 증가시키는 함수 구현하기

- 정수형 변수에 대해 자신의 값을 1만큼 증가시키려고 사용하는 연산자가 ++다. ++ 연산자는 선행처리와 후행처리, 2가지 형태로 사용된다.

```
int a=10, b=10, c;
c=++a;                          // 선행처리
cout<<a<<" "<<c<<endl;          // 11 11
c=b++;                          // 후행 처리
cout<<b<<" "<<c<<endl;          // 11 10
```

- 선행처리 방식으로 피연산자를 1만큼 증가하는 AddOnePrefix를 멤버함수로 정의해보자.

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

- ++ 연산자를 AddOnePrefix 함수로 멤버함수로 구현하면 포인터 this가 있으므로 매개변수를 지정하지 않아도 된다.

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

06 객체를 다루기 위한 함수

- 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. 그래서 this 앞에 * 연산자를 붙여주어야 하므로 addOne Prefix 함수가 반환할 값은 *this 형태이어야 한다.

```
return *this;
```

- 후행처리하는 ++ 연산자와 동일한 동작을 하는 AddOnePostfix 함수를 정의해 보자.

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

- 후행처리하는 ++ 연산자는 선행처리하는 ++ 연산자처럼 자신의 값을 1만큼 증가시키는 용도로도 쓰이지만 다음처럼 대입 연산자와 같이 사용하면 증가되기 전의 현재 상태 값을 c에 우선 저장한 후 피연산자인 b의 값이 1만큼 증가하게 된다.

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

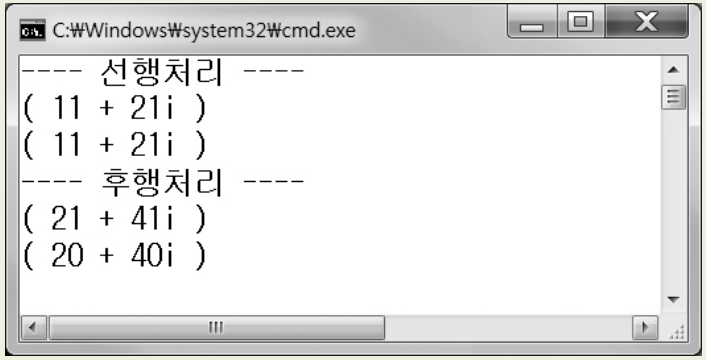
06 객체를 다루기 위한 함수

- 함수 AddOnePostfix도 함수명 앞에 붙은 객체 y에 대한 정보를 포인터 this가 받아온다. 후행처리는 피연산자 y가 증가되기 전의 값이 함수의 결과값으로 반환되어야 하므로 this로 접근해서 피연산자 y값을 증가하기 전에 객체 y의 값을 임시 기억장소인 temp에 저장해 둔다. 포인터 this로 접근해서 real, image 멤버의 값을 1만큼 증가시킨 후 결과값을 반환할 때는 증가되기 전의 값(temp)을 반환한다.

- 객체 자신의 값을 1만큼 증가시키는 함수를 프렌드 함수로 만들어 보자.
- 일반함수는 main에 선언된 객체를 매개변수로 넘겨주어야 하는데 main 함수에 선언된 객체의 값을 1만큼 증가시켜야 하므로 매개변수로 객체 값을 변경할 수 있어야 한다. 따라서 '주소에 의한 전달 방식'이나 '참조에 의한 전달 방식'으로 구현해야 한다.
- '주소에 의한 전달 방식'으로 구현하면 호출할 때 피연산자로 사용되는 객체 변수의 주소값을 함수의 실 매개변수로 지정해야 하므로 매개변수를 참조 변수로 작성하는 '참조에 의한 전달 방식'으로 구현해 보자.

예제 11-19. 자신의 값을 1만큼 증가시키는 프렌드 함수 작성하기(11_19.cpp)

책의 소스코드 참고



07 연산자 오버로딩

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
}
```

Complex 객체를 피연산자로 하는 연산자 오버로딩

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

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

07 연산자 오버로딩

- 두 객체를 더하려고 기술한 z = x + y; 문장은 프로그래머에게 보기 편한 형태지만 연산자 함수를 쉽게 정의하려고 컴파일러에 의해서는 호출되는 형태로 변경해 보자.

  z = x.operator+(y);

- 복소수의 덧셈을 구하는 멤버함수 Sum을 호출하는 구조와 동일하다.

  z = x.Sum(y);

- + 연산자 함수의 정의 역시 Sum 함수와 동일하다. 단지 함수명만 Sum에서 operator로 바꾸어 놓은 것 같다.

함수로 구현한 예	연산자로 구현한 예

Complex Complex::**Sum(Complex**
rightHand)
{
...
}
	Complex Complex::**operator+(Complex**
	rightHand)
	{
	...
	}

- Complex 클래스에 대해서도 2가지 용도로 정의할 수 있는 - 연산자를 오버로딩해 보자. 먼저 뺄셈 용도로 사용되는 - 연산자부터 정의해 보자.

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

07 연산자 오버로딩

- 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.

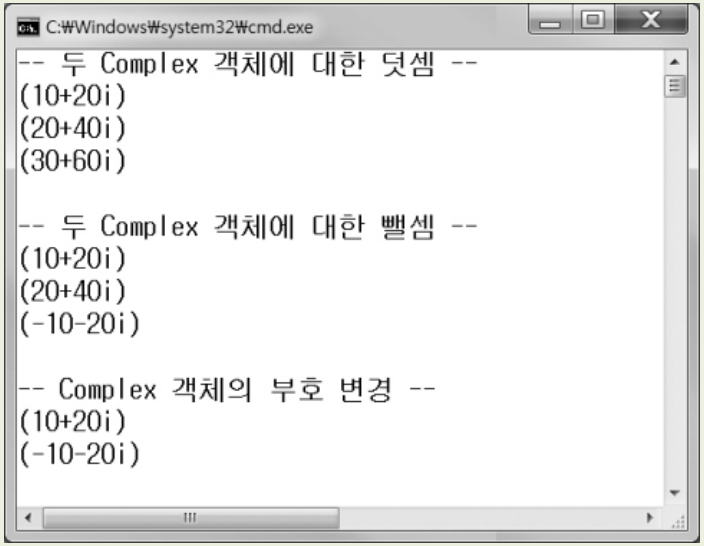
```
z = x - y ; // z = x.operator-(y);
```

- 뺄셈이 아닌 음수 기호로서의 - 연산자를 정의해 보자. 변수 앞에 – 연산자를 기술하면 부호는 바뀌지만 변수는 바뀌지 않는다. 클래스 Complex도 객체의 부호를 바꿀 수 있도록 – 연산자를 오버로딩해야 한다.

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

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

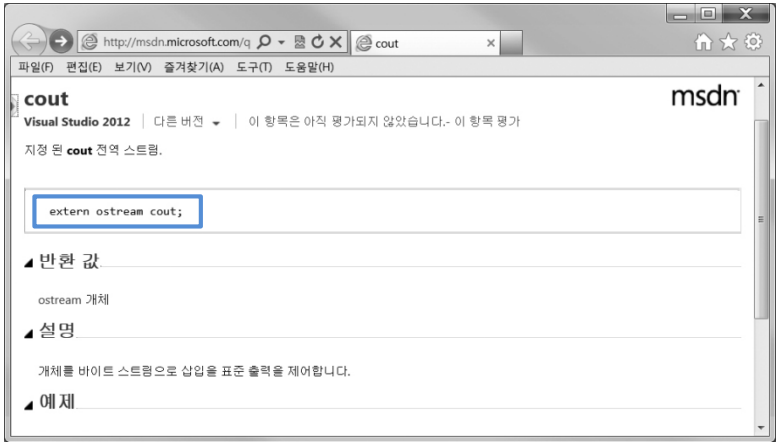
책의 소스코드 참고



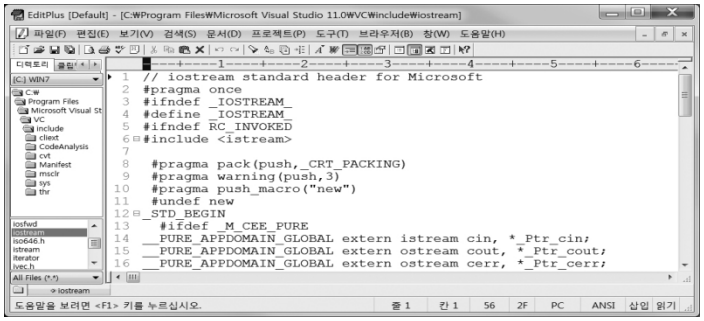
07 연산자 오버로딩

출력을 담당하는 연산자 오버로딩

- cout을 블록으로 지정하고 <F1> 키를 누르면 다음과 같은 내용을 발견할 수 있다.



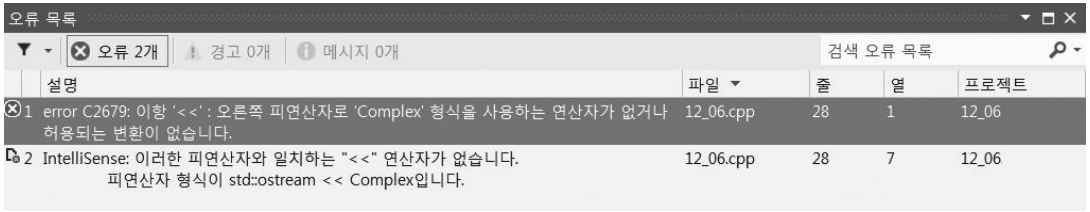
- cout은 주로 화면과 같은 표준 출력장치에 출력을 담당하는 객체다. cout 객체를 사용하려면 현재 작업 중인 소스파일에 iostream.h 헤더파일을 반드시 포함해야 한다. cout 객체가 iostream.h에 정의되어 있기 때문이다.



07 연산자 오버로딩

- ShowComplex 함수 대신 cout 객체에 << 연산자를 사용해서 출력하려고 하면 컴파일 에러가 발생한다.

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



- cout 객체가 Complex 객체를 출력할 수 있도록 하려면 ostream 클래스에 매개변수가 Complex 클래스인 operator<< 연산자 함수를 오버로딩하면 된다.
- 연산자를 오버로딩하는 방법에는 2가지가 있다. 하나는 멤버함수를 사용하는 방법이고 다른 하나는 일반 함수(프렌드 함수)를 사용하는 방법이다.
- Complex 객체를 출력하는 문장을 연산자 함수 형태로 표현해 보자.

연산자 형태	연산자 함수 형태(멤버함수로 구현할 경우)
cout << x	cout.operator<<(x)

- ostream은 C++에서 제공하는 클래스이므로 이 클래스를 프로그래머가 함부로 바꾸는 것은 바람직하지 않다. 그리고 클래스 Complex에 오버로딩하면 연산자 함수를 호출할 때 Complex 객체를 연산자 함수 앞에 붙여야 한다.

```
x.operator(cout);
```

07 연산자 오버로딩

- 연산자 형태로 바꾸어보면 다음과 같다.

```
x << cout;
```

- 기본 자료형 출력을 하려고 cout<<x와 같이 표현했는데, Complex 객체를 출력할 때마다 두 피연산자의 위치가 바뀐다면 혼동이 될 것이다.
- << 연산자가 일반 함수 형태로 정의되어 있다면 이를 컴파일러가 어떻게 해석할지 살펴보자.

연산자 형태	연산자 함수 형태(멤버함수로 구현할 경우)
cout << x	operator<<(cout, x);

```
void operator<<(ostream &os, const Complex &comObj)
{
os<<"( " <<comObj.real <<" + " <<comObj.image << "i )"
<<endl ;
}
```

- 앞선 예의 출력을 위한 << 연산자의 오버로딩은 완벽하지 않다. 왜냐하면 기본 자료형을 출력할 때는 << 연산자가 연속적으로 사용되기 때문이다.

```
int a = 5;
int b = 10;
cout<< a << " + " << b;
```

07 연산자 오버로딩

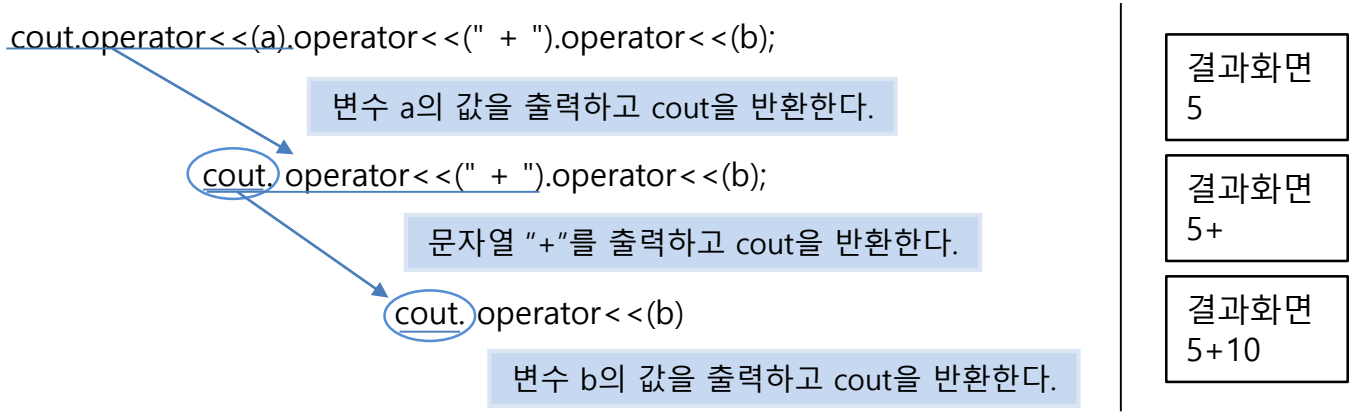
- cout 객체는 맨 왼쪽에 한번 기술하고 출력할 대상의 자료형이 바뀔 때마다 << 연산자를 연속적으로 기술한다. 이렇게 연속적으로 기술할 수 있는 이유가 무엇인지 알아보기 위해서 iostream.h 헤더파일에 기본 자료형을 피연산자로 하는 << 연산자가 어떻게 기술되어 있는지 살펴보자.

```
ostream & operator<<(int);
ostream & operator<<(char *);
```

- 함수의 자료형이 ostream &로 선언되어 있는데, 이는 cout 객체에 << 연산자를 연속적으로 기술해서 자료 여러 개를 출력할 수 있도록 하기 위해서다.

```
cout<< a << " + " << b << endl;
```

- 이 문장을 컴파일러가 해석하는 문장으로 해석해 보면 다음과 같다.



07 연산자 오버로딩

- Complex 객체를 매개변수로 하는 << 연산자도 Complex 객체를 출력하고 다른 자료를 연속적으로 출력하려면 반환값이 ostream &로 선언되어야 한다.

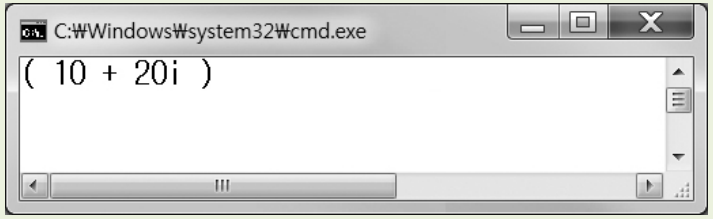
```
ostream &operator<<(ostream &os, const Complex &comObj)
{
os<<"( " <<comObj.real <<" + " <<comObj.image << "i )" <<endl ;
return os;
}
```

- 일반 함수 형태로 << 연산자를 오버로딩하면 내부에서 private 멤버인 real과 image에서 컴파일 에러가 발생한다. 그러므로 Complex 클래스 내부에 << 연산자를 프렌드 함수로 선언해야 한다.

```
class Complex{
public :
friend ostream & operator<<(ostream &os , Complex &comObj);
}
```

예제 11-21. 출력을 위한 << 연산자 오버로딩하기(11_21.cpp)

책의 소스코드 참고



07 연산자 오버로딩

연산자를 오버로딩할 때의 주의사항

❶ C++에서 이미 사용하던 연산자만 오버로딩할 수 있다. $ 기호는 C++에서 연산자로 사용되지 않는다. 그러므로 operator$( ) 함수를 정의할 수 없다.

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

```
int a;
% a; // 나머지를 구하는 연산자를 단항 연산자로 사용하지 못한다.
Complex x;
% x; // 나머지를 구하는 연산자를 단항 연산자로 오버로딩할 수 없다.
```

❸ C++에서 사용하는 연산자 중에서 다음 연산자는 오버로딩할 수 없다. .(멤버 참조 연산자), ::(스코프 연산자), ?:(조건 연산자), sizeof(sizeof 연산자), *(포인터 연산자)

❹ 연산자를 오버로딩하려면 피연산자가 적어도 하나 이상은 사용자 정의 자료형이어야 한다. 이는 기본 자료형에 대한 연산자 오버로딩을 방지하기 위해서다. 다음은 잘못된 예다.

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

❺ 대부분의 연산자는 멤버함수 또는 프렌드 함수로 오버로딩할 수 있다. 그러나 다음 연산자는 멤버함수로만 오버로딩할 수 있다.

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