Chapter

10

클래스와 객체

* 학습목표

- 객체지향 프로그래밍에 대해서 이해한다.
- 클래스를 정의하고 객체를 생성해서 사용하는 방법을 익힌다.
- 클래스 멤버인 멤버변수와 멤버함수에 대해서 살펴본다.
- 캡슐화를 위한 접근지정자에 대해서 학습한다.

Section

01 클래스의 이해

C++는 C 언어에 객체지향 프로그래밍(Object Oriented Programming)의 개념을 도입한 것이다. 지금까지는 일반적인 프로그래밍 언어에서 공통적으로 꼭 알아야 할 사항을 학습했다. 이제 C++의 특징이라고 할 수 있는 객체지향 프로그래밍에 대해 본격적으로 학습할 차례다.

1 클래스의 선언

객체지향 프로그램은 우선 클래스를 선언한 후 이를 인스턴스(객체)화해서 프로그램을 작성한다. C++에서 클래스를 선언하는 예약어는 class다. class는 자료를 추상화해서 사용자 정의 자료형으로 구현할 수 있게 하는 C++의 도구다. class로 클래스를 선언하는 형식은 구조체와 유사하지만 개념은 확연히 다르다. 구조체는 단순히 멤버변수들을 정의해서 원하는 형태로 기억공간을 할당받을 수 있도록 정의하는 것이지만 클래스는 기억공간을 확보하는 것은 물론이고 이 클래스를 다룰 수 있는 방법도 구현해야 한다. 그러므로 클래스는 기억공간을 확보하는 멤버변수는 물론 멤버변수를 다루기 위한 멤버함수도 함께 정의해야 한다.

클래스 선언	클래스 멤버함수 정의

class 클래스명 {
  접근 지정자 :
  자료형 멤버변수; → 멤버변수 선언
 	
  접근 지정자 :
  자료형 멤버함수(); → 멤버변수 원형 정의
};
	자료형 클래스명::멤버함수()
	{
	  ...
	  ...
	}

클래스는 크게 클래스 선언과 클래스 멤버함수 정의로 구성되어 있다. 클래스 선언에는 멤버변수와 멤버함수 원형을 정의한다. 멤버함수 정의는 클래스 선언 밖에서 따로 이루어진다.

클래스 선언은 구조체 선언과 그 구조가 비슷하지만 구조체에는 없었던 접근 지정자가 추가된다. 접근 지정자는 클래스에서 각 멤버변수나 멤버함수 앞에 붙여 각각에 대한 접근 권한을 지정해주는데, 접근 권한 예는 private, public, protected 3가지가 있다. protected는 상속성과 연관된 접근 지정자이므로 상속성을 공부하는 과정에서 자세히 다루도록 하고 우선 [표 10-1]의 private와 public을 살펴보자.

[표 10-1] 접근 지정자

구분	현재 클래스 내	현재 클래스 밖
private	o	x
public	o	o

private로 선언된 멤버는 해당 멤버가 속한 클래스의 멤버함수에서만 사용할 수 있기 때문에 private로 선언된 멤버는 캡슐화(데이터 은닉)된다.

public으로 선언된 멤버들은 객체를 사용할 수 있는 범위라면 어디서나 접근이 가능한 공개된 멤버다. 주로 private 멤버를 해당 클래스 외부에서 사용하도록 하기 위한 멤버함수를 정의할 때 사용한다.

이 두 접근 지정자에 대해 좀 더 구체적으로 알아보려고 수학에서 사용하는 복소수를 Complex라는 이름의 클래스로 설계해 보자. 클래스를 설계하기에 앞서 복소수(複素數, complex number)에 대한 이해가 필요한데, 복소수는 실수와 허수의 합으로 나타내는 수다. a, b를 실수, i를 허수라고 할 때, 복소수는 a+bi로 나타낸다. 그리고 이때 a를 실수부, bi를 허수부라고 한다.

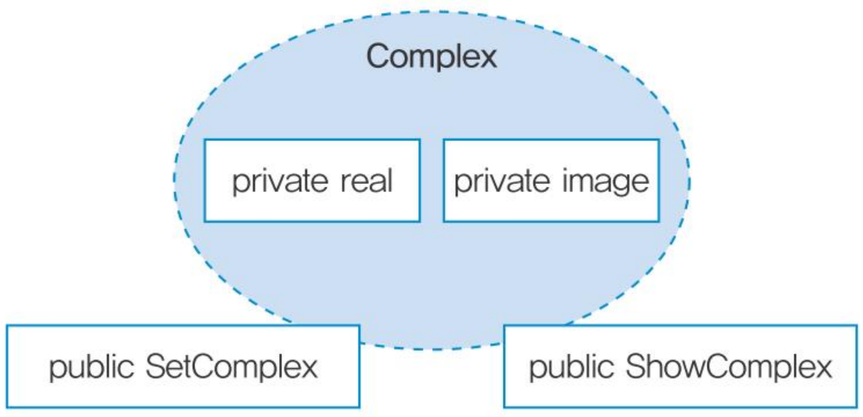
클래스를 새로운 자료형으로 설계할 때에는 단순히 데이터의 저장이라는 측면뿐만 아니라 데이터를 처리할 함수(멤버함수) 제공도 고려해야 한다. 그러므로 다음과 같이 클래스는 새로운 자료형이 되고 데이터는 멤버변수가, 데이터를 처리할 함수는 멤버함수가 된다.

- 새로운 자료형(클래스) = 데이터의 저장(멤버변수) + 데이터를 처리할 함수(멤버함수)

이를 바탕으로 이제 복소수에 대한 정보를 저장하고 이를 처리하기 위한 클래스를 구현해 보자. 우선 실부와 허수를 저장해야 하므로 이를 멤버변수로 구현한다. 실수를 저장하기 위한 멤버변수를 real, 허수를 저장하기 위한 멤버변수를 image라고 하자.

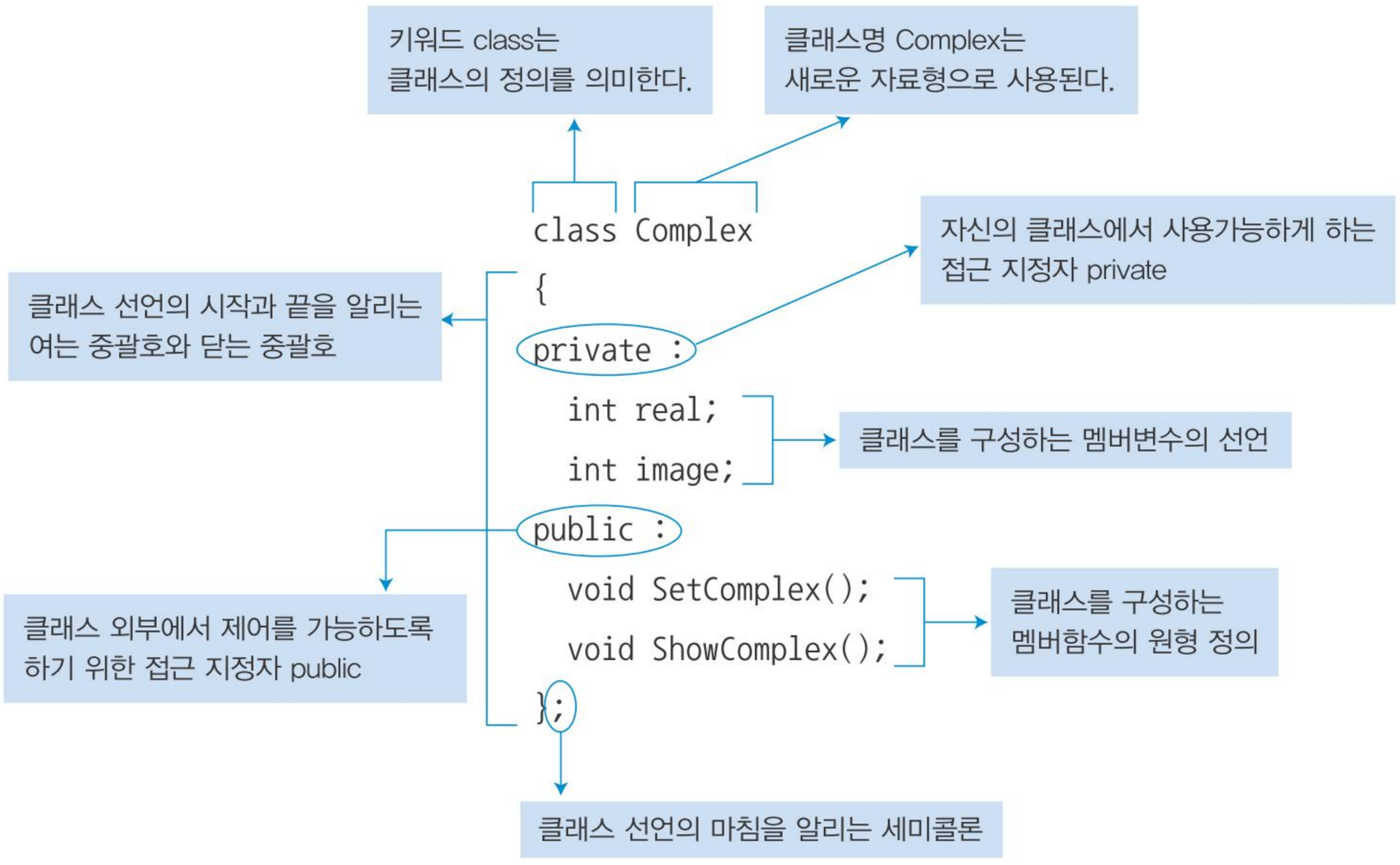
클래스를 구성하는 멤버들은 그것이 변수든 함수든 상관없이 public 또는 private로 선언할 수 있다. 하지만 데이터 은닉에 입각해서 멤버변수의 접근 지정자는 private로 선언하는 것이 일반적이다. private로 선언된 멤버변수는 직접 제어하지 못하므로 이를 다룰 수 있도록 제공하는 멤버함수는 클래스 외부에서 접근 가능하도록 접근 지정자를 public으로 선언하는 것이 일반적이다.

그러므로 여기서도 [그림 10-1]과 같이 두 멤버변수에 값을 설정(SetComplex)하거나 알아내기(ShowComplex) 위한 멤버함수를 클래스 외부에서 접근할 수 있도록 공개한다. 이런 데이터의 은닉은 데이터를 직접 접근하지 못하게 하는 보호 측면도 있지만 프로그래머가 객체의 세부사항을 모두 알아야 할 필요 없이 멤버함수를 사용해서 원하는 작업을 할 수 있는 편리함도 제공한다.



[그림 10-1] Complex 클래스 설계

위에서 설계한 복소수 클래스를 직접 구현해보면 다음과 같다. 여기서는 클래스의 선언부만 살펴보는데, 클래스의 선언은 멤버변수의 선언과 멤버함수의 선언으로 구성된다.



[그림 10-2] Complex 클래스 구현

클래스의 멤버함수 구현

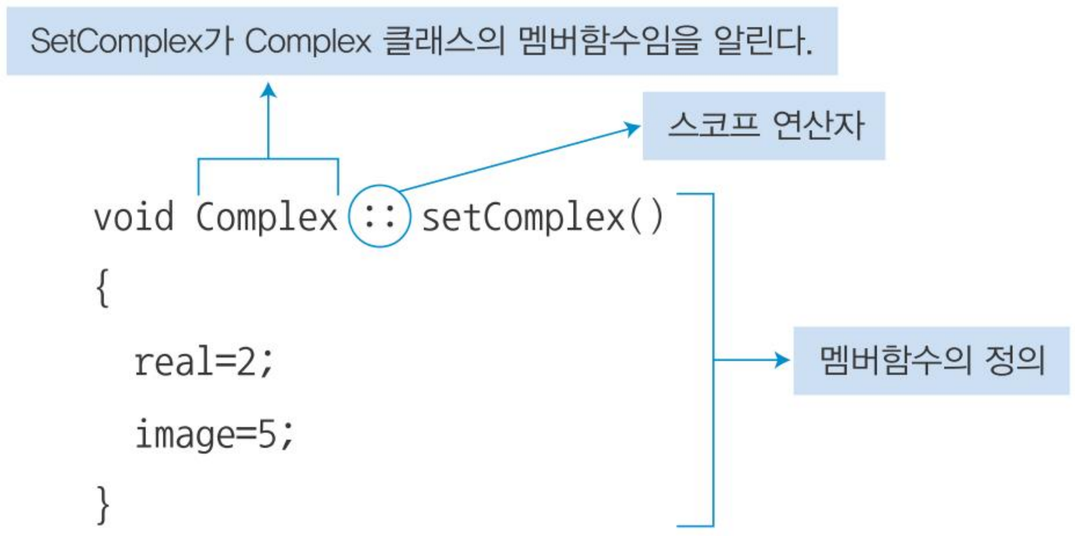
함수의 3요소는 함수의 정의, 선언, 호출이다. 멤버함수도 이 3가지 조건을 만족해야 한다. 클래스 선언에서는 멤버함수의 선언만을 기술했다. 멤버함수의 호출은 일반 함수와 달리 클래스로 객체를 선언해서 인스턴스화가 되어야 호출할 수 있으므로 객체를 선언하는 방법을 설명할 때 멤버함수를 호출하는 방법을 알아볼 것이므로 우선 함수의 구성요소 중 나머지 한 개인 멤버함수의 정의부터 알아보자.

멤버함수의 정의는 함수의 머리와 함수의 몸체로 구성되고 함수의 자료형(반환값)과 함수의 매개변수도 있다는 면에서 일반 함수의 정의와 비슷하다. 하지만 다음과 같이 멤버함수만의 몇 가지 특징이 있다.

❶ 멤버함수를 정의할 때 그 함수가 어느 클래스에 소속되는지 나타내려면 함수명 앞에 클래스명을 명시해야 한다. 이때 클래스명 다음에 스코프 연산자(::)를 사용한다.

❷ 클래스의 멤버함수를 정의하는 목적은 클래스 내에 정의된 private 멤버에 접근해서 이를 다루기 위해서다. 그래서 멤버함수를 '메소드'라고도 한다.

다음은 Complex 클래스에 선언된 멤버함수 중 SetComplex에 대한 정의다. SetComplex 함수는 실수부와 허수부에 각각 값을 부여한다.



[그림 10-3] SetComplex에 대한 정의

함수의 머리 부분에는 어느 클래스에 소속되어 있는지를 식별하기 위해 함수명 앞에 클래스명과 함께 스코프 연산자(::)를 사용해야 한다. SetComplex 함수 앞에 기술한 Complex::는 해당 함수가 Complex 클래스에 소속된 멤버함수라는 의미다.

만약 함수를 정의할 때 아래 ❶과 같이 정의되어 있다면 소속된 클래스명이 명시되어 있지 않기 때문에 이 함수는 일반 함수로 인식한다. 그래서 일반 함수에서는 선언하지 않은 real과 image라는 변수는 사용할 수 없으므로 컴파일 에러가 발생한다.

멤버함수를 정의할 때 클래스명을 명시하는 것은 어느 클래스에 소속되어 있는지를 식별하기 위해서지만 다른 클래스에서 같은 이름의 멤버함수를 사용할 수 있게 해주려는 것이기도 하다. 그러므로 아래 ❷와 같이 명시하면 SetComplex라는 이름의 함수를 클래스 Test의 멤버함수로 사용할 수도 있다. 이름이 같은 함수를 여러 번 정의할 수 있는 이유는 스코프 연산자 앞에 어느 클래스의 소속인지를 명시하기 때문이다.

멤버함수의 또 다른 특징은 클래스의 private 멤버에 접근할 수 있도록 한다는 것이다. ❸은 SetComplex 함수의 내용이다.

❶
```
void SetComplex()
{
  real=2;
  image=5;
}
```

❷
```
void Test::SetComplex()
{
   . . . . .
}
```

❸
```
void complex::SetComplex()
{
  real=2;
  image=5;
}
```

real과 image는 Complex 클래스의 private 멤버변수다. private 멤버변수는 자신이 속한 클래스의 멤버함수를 제외하고는 다른 어떤 함수에서도 접근할 수 없다. 멤버함수 이외의 함수에서 접근하면 에러가 발생한다. 만일 멤버함수가 없다면 private 멤버변수는

외부에서 접근할 수 없게 되므로 무용지물이 된다. 그렇다고 멤버변수를 public으로 선언하면 이는 객체지향 개념에서 벗어난 클래스 설계가 된다. private 멤버변수를 정의했다면 이를 다룰 수 있는 방법도 제공해야 하는데, 이것이 바로 멤버함수의 기능이다. 멤버함수는 private 멤버변수에 대한 처리를 하려고 구현하는 함수다.

2. 객체 선언과 멤버 참조

"사람에게 커피 한 잔을 대접하세요."라고 하면 도대체 어떤 사람을 지칭하는지 황당할 것이다. 반면 "성윤정씨에게 커피 한 잔을 대접하세요."라고 하면 구체적인 인물인 '성윤정'을 지칭했으므로 커피를 당장 대령할 것이다. 여기서 '사람'은 구체적이지 않고 추상화된 개념이기 때문이다. 이와 같이 클래스는 추상적인 개념이고 객체는 '성윤정'과 같이 구체적인 실체다. 그러므로 객체지향 언어에서는 클래스와 객체에 대한 개념을 올바르게 구별할 수 있어야 한다.

이제 본격적으로 C++ 문법으로서의 클래스와 객체를 살펴보자. 클래스는 추상화된 자료형이므로 클래스로는 프로그램 내에서 실질적인 작업을 수행하지 못한다. 따라서 객체를 선언해야 하는데, 객체는 클래스를 실체(instance)화한 것이다. 그래서 객체를 '인스턴스'라고도 한다. 클래스와 객체의 개념을 좀 더 쉽게 이해하려고 기본 자료형을 예로 들어보자. 다음은 변수를 선언한 후 변수에 5라는 값을 저장하는 예다.

```
int a;  // int형의 변수 a를 선언한다.
a=5;    // 변수 a에 값 5를 대입한다.
```

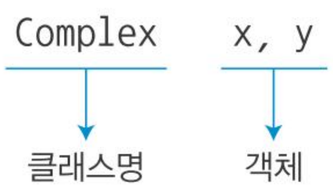
정수형 자료형인 int로 a라는 변수를 선언한 후 변수 a에 5라는 값을 저장한다. 이때 자료형 int가 클래스에 해당되고 변수명 a가 바로 객체에 해당된다. 프로그램을 진행하는 동안에는 변수 a가 사용되는데, 이렇게 하려면 자료형 int로 변수 a를 선언해야만 한다. 자료형 int는 변수 a를 만들어내기 위한 형틀 즉, 템플릿(template)이다.

클래스도 객체를 생성해내기 위한 일종의 형틀로 이해하면 쉽다. 다음은 객체를 선언하는 기본 형식이다. 객체 선언은 변수 선언과 동일하다. 자료형 위치에 클래스명을 기술하고 변수명 위치에 객체명을 선언하면 된다.

객체 선언 기본 형식

```
[class] 클래스명 객체명1, 객체명2, ..., 객체명n;
```

다음은 앞서 정의해 둔 Complex 클래스를 이용해서 x와 y라는 객체 2개를 생성한 예다. 객체 x, y는 멤버변수 real과 image를 포함하고 있는데, 이 두 멤버변수는 private로 선언되어 있으므로 이를 사용하려고 SetComplex와 ShowComplex를 멤버함수로 제공한다.



3. 클래스 멤버의 접근 방법

클래스는 구조체처럼 멤버들로 구성된 집합체다. 클래스로 객체 선언을 마쳤다면 프로그램을 작성하는 동안에는 객체를 직접 사용하기보다는 객체를 구성하는 멤버를 사용해야 한다. 멤버를 사용하려면 구조체처럼 .과 -> 같은 멤버참조 연산자를 사용한다. -> 연산자는 객체 포인터를 공부한 후 사용하기로 하고 여기서는 객체변수를 선언해서 . 연산자로 멤버를 참조하는 방법을 학습한다.

. 연산자를 이용한 클래스 멤버 접근 방법

```
객체명.멤버변수;
객체명.멤버함수();
```

저자 한마디

클래스에서 범하기 쉬운 실수

'객체'를 좀 더 쉽게 설명하면 '클래스로 선언된 변수'라 할 수 있다. 아마 다음과 같이 작성하는 사람은 없을 것이다. 이는 분명히 잘못된 문장이기 때문이다.

```
int=5;  // 에러
```

그런데 Complex가 클래스라면서 다음과 같이 어처구니 없는 문장을 작성하는 경우가 있다.

```
Complex.real = 2;
```

int=5가 잘못된 문장이듯이 Complex라는 클래스를 직접 사용하는 것도 잘못된 문장임을 명심해야 한다. Complex는 메모리 할당을 받은 실체가 아니고 클래스는 객체를 선언하기 위한 형틀로서 의미가 있으므로 클래스는 객체 선언문에서 자료형 위치에 기술할 경우에만 유용하다.

다음은 복소수를 구현한 Complex 클래스로 객체를 생성한 후 멤버함수를 호출하는 예다.

```
Complex x, y;
x.SetComplex();
y.SetComplex();
```

객체 x와 y의 멤버변수 real, image는 private로 선언되어 있으므로 직접 접근해서 사용하지 못한다. 멤버함수 SetComplex를 public으로 선언하고 이를 통해서 객체에 접근하도록 했다. 주의할 것은 SetComplex가 멤버함수이므로 함수명만으로 호출할 수 없다는 것이다. 반드시 함수명 앞에 객체명과 멤버참조 연산자를 함께 기술해야 한다.

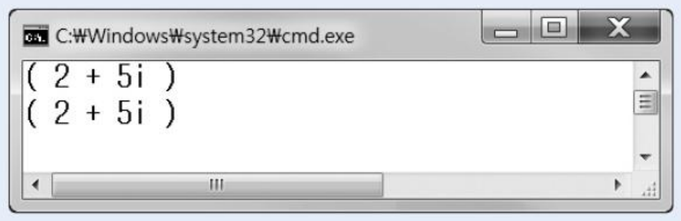
```
SetComplex();    // 에러 ------------------ 잘못된 예(×)
x.SetComplex(); // 객체명.멤버함수(); ------ 올바른 예(○)
```

C 언어만 접해 보았다면 함수를 호출할 때 객체명을 붙이는 것이 어색할 수 있지만 비주얼 베이직이나 기타 다른 객체지향 언어를 접해 보았다면 멤버함수 앞에 객체명을 붙이는 것이 낯설지 않을 것이다.

[예제 10-1]은 복소수를 클래스로 설계한 후 그 클래스로 객체를 인스턴스화해 보는 프로그램이다.

예제 10-1 복소수를 클래스로 설계하기(10_01.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     void SetComplex();
10     void ShowComplex();
11 };
12
13 void Complex::SetComplex()
14 {
15   real=2;
16   image=5;
```



```
17 }
18 void Complex::ShowComplex()
19 {
20   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
21 }
22 void main()
23 {
24   Complex x, y;
25
26   x.SetComplex();
27   x.ShowComplex();
28   y.SetComplex();
29   y.ShowComplex();
30 }
```

03행~11행 클래스 선언부로서, 멤버변수의 선언과 멤버함수의 선언으로 구성되어 있다. 멤버변수의 선언을 하기에 앞서 05행에서 접근 지정자를 private로 주었다. private:라고 접근 지정자를 적어주면 public:과 같은 다른 접근 지정자를 만나기 전까지는 그 지정자의 영향을 받기 때문에 두 접근 지정자 사이의 모든 멤버에 private가 적용된다. 여기서는 real과 image가 모두 private 멤버변수가 된다.

08행~10행 멤버함수를 선언하기에 앞서 08행에서 접근 지정자를 public으로 주었다. public: 이후에 다른 접근 지정자가 기술되지 않았으므로 SetComplex 함수와 ShowComplex 함수는 모두 public 멤버가 된다.

13행~21행 11행에서 클래스의 선언을 종료한 후 클래스 외부에 멤버함수를 정의했다. 이때 주의할 점은 어느 클래스의 멤버함수인지를 식별하려고 클래스명을 스코프 연산자(::)와 함께 명시해야 한다는 것이다.

24행 Complex 클래스를 선언한 후에는 객체화하는 작업을 해줘야 하는데, 여기서는 main 함수에서 기본 자료형을 선언하는 방법과 동일하게 객체를 선언했다.

26행~29행 24행의 객체 선언에 의해 x와 y가 인스턴스화되었으므로 멤버들을 객체로 접근해서 사용했다. 객체가 선언되었다고 모든 멤버를 다 사용할 수 있는 것은 아니다. public 멤버만이 사용할 수 있고 private 멤버는 객체로 접근해서 사용할 수 없다. SetComplex와 ShowComplex 함수의 접근 지정자가 public이므로 이들 함수만 main 함수에서 사용했다.

4. 클래스의 접근 지정자, private/public

클래스는 데이터의 은닉이라는 객체지향 개념을 구현하기 위해 접근 지정자로 private를 추가했다. 그리고 private 접근 지정자와 개념이 반대인 public 접근 지정자도 존재한다. 이들 접근 지정자에 대해 완벽히 이해해야만 객체지향의 개념을 이해한다고 할 수 있으므로 우선 접근 지정자를 다시 한 번 정리해 보고 예제를 통해 이 둘의 차이점을 살펴보자.

private 접근 지정자

❶ 접근 지정자가 생략되면 기본(default)으로 private가 적용된다. 하지만 private:를 명시적으로 기술할 수도 있다.

❷ private 멤버의 사용 범위는 소속된 클래스 내의 멤버함수로 국한된다.

❸ 일반적으로 멤버변수를 private로 설정한다.

public 접근 지정자

❶ public 멤버로 지정하려면 public:을 명시적으로 기술해야 한다.

❷ 클래스 내의 멤버함수에서는 물론 객체가 선언되어 있는 영역이라면 어디서든지 객체명 다음에 멤버참조 연산자(.)로 연결해서 멤버함수를 사용할 수 있다.

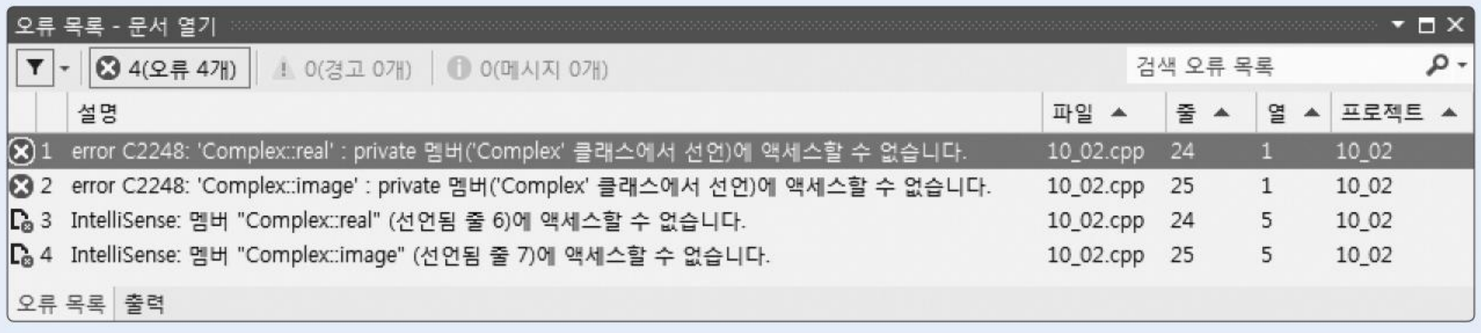
❸ private 멤버변수를 처리하기 위한 목적으로 작성하는 멤버함수는 일반적으로 public 멤버로 설정한다.

[예제 10-2]를 실행시키면 에러가 발생한다. 이 예제의 실행 결과에서 발생하는 에러 메시지를 통해 에러가 발생하는 이유를 살펴보고, private 접근 지정자를 사용할 때 에러가 발생하지 않게 하는 방법을 알아보자.

예제 10-2 private 멤버 성격 파악하기(10_02.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     void SetComplex();
10     void ShowComplex();
11 };
12 void Complex::SetComplex()
13 {
14   real=2;
15   image=5;
16 }
17 void Complex::ShowComplex()
```

```
18 {
19   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
20 }
21 void main()
22 {
23   Complex x;
24   x.real = 5 ;    // 컴파일 에러
25   x.image = 10;  // 컴파일 에러
26   x.ShowComplex();
27 }
```



24행~25행 private 멤버변수인 real과 image는 main 함수에서 사용할 수 없으므로 컴파일 에러가 발생한다. private 멤버는 반드시 Complex 클래스 내부에 선언된 멤버함수에서만 사용할 수 있다.

12행~20행 Complex 클래스의 멤버함수에서 real과 image를 사용했기 때문에 이곳에서는 에러가 발생하지 않는다.

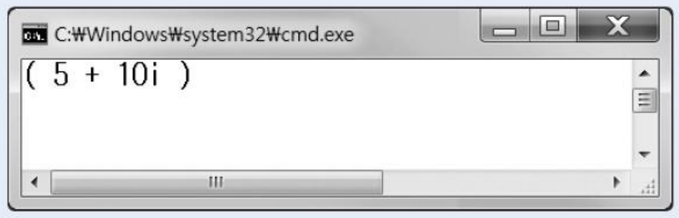
그럼 이 예제의 에러를 해결하는 방법을 알아보자. 크게 다음 2가지 방법을 사용할 수 있다.

❶ main 함수에서 real과 image를 사용할 수 있도록 이 두 멤버변수의 접근 지정자를 public으로 변경한다.

❷ 위의 방법은 문제를 간단히 해결하지만 private 접근 지정자를 public으로 변경하면 데이터 은닉이라는 객체지향 개념을 위배하므로 최선의 방법이 아니다. C++가 나오게 된 배경이 객체지향 개념이므로 데이터 은닉이라는 특성을 지키려면 멤버변수의 접근 지정자는 private로 유지해야 한다. 그럼 private 멤버변수의 값을 변경하려면 어떻게 해야 할까? 멤버변수에 직접 접근할 수 없으므로 이런 작업을 할 수 있도록 멤버함수를 추가하면 된다. 멤버변수의 값을 변경하는 로직을 멤버함수에 기술하고 특정 멤버변수의 값을 변경하기 위해 멤버함수를 호출하면 된다.

[예제 10-3]은 멤버함수를 추가해서 [예제 10-2]의 문제를 해결한 프로그램이다.

예제 10-3 private 멤버를 다루기 위한 멤버함수 추가하기(10_03.cpp)




```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09   void SetComplex();
10   void ShowComplex();
11   void SetReal(int r);
12   void SetImage(int i);
13 };
14
15 void Complex::SetComplex()
16 {
17   real=2;
18   image=5;
19 }
20 void Complex::ShowComplex()
21 {
22   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
23 }
24 void Complex::SetReal(int r)
25 {
26   real = r;
27 }
28 void Complex::SetImage(int i)
29 {
30   image = i;
31 }
32
33 void main()
34 {
35   Complex x;
36   x.SetReal( 5 );
37   x.SetImage( 10 );
38   x.ShowComplex();
39 }
```

36행~37행 실수부와 허수부 값을 저장하는 설정을 하려고 하는데, 멤버변수가 private로 선언되어서 이를 직접 사용하지 못한다. 그러므로 실수부에 값을 설정하려면 멤버함수 SetReal을 호출하고, 허수부에 값을 설정하려면 멤버함수 SetImage를 호출해야 한다.

11행~12행 실수부와 허수부를 설정하기 위한 SetReal 함수와 SetImage 함수는 클래스 Complex에 멤버함수로 선언되어 있어야 한다.

24행~27행 실수부의 값을 설정하는 SetReal 함수를 정의했다. SetReal 함수를 호출할 때 넘겨준 값을 멤버함수의 형식 매개변수인 r이 전달받아 실수부를 저장하는 private 멤버변수인 real에 대입한다.

28행~31행 허수부의 값을 설정하는 SetImage 함수를 정의했다. SetImage 함수는 매개변수로 넘겨받은 값을 허수부 멤버변수 image에 대입한다.

5_ 클래스 내부에 멤버함수 정의하기

멤버함수의 정의가 아주 짧으면 클래스 선언 내부에 직접 정의할 수 있다. 클래스 내부에 함수를 정의하기에 앞서 우선 인라인 함수에 대해 다시 살펴보자.

인라인 함수

9장에서도 알아보았듯이, 인라인 함수는 매크로 함수와 실행 원리가 동일하다. 즉, 함수가 호출될 때 그 위치에 해당 함수가 삽입되도록 하는 것이다. 그래서 함수를 호출할 때 생기는 시간 지연을 줄일 수 있다는 장점이 있다. 프로그램 속도를 높이려고 보강한 인라인 함수는 함수가 실제적으로 호출되지 않고 프로그램 코드 사이에 컴파일된 함수 코드가 삽입된다. 그러나 인라인 함수를 10번 호출하면 함수의 복사본 10개가 생기므로 함수가 긴 경우에는 프로그램 코드가 그만큼 길어져서 프로그램이 커진다는 단점이 있다. 그러므로 인라인 함수는 주로 정의가 짧을 때 사용하고, 인라인 함수를 정의할 때는 함수 선언 앞에 inline이라는 예약어를 써 준다.

인라인 함수 기본 형식

```
inline 자료형 함수명 (매개변수리스트)
{
  변수 선언;
  문장;
  return (결과값);
}
```

9장에서도 언급했듯이, 매크로 함수가 선행처리기에 의해 매크로 확장을 하면서 단순 치환하기 때문에 괄호 연산자를 확실히 기술하지 않으면 오류가 생긴다. 그리고 이러한 오류를 방지할 수 있는 좀 더 안정된 함수가 인라인 함수다. 인라인 함수는 동작원리가 매크로 함수와 동일하지만 함수를 정의하는 방식은 일반 함수와 같아 부작용이 생기지 않는다. 또한 함수의 매개변수나 반환값에 대한 자료형이 명시되어 있으므로 매크로 함수보다는 함수의 구조가 정교하다.

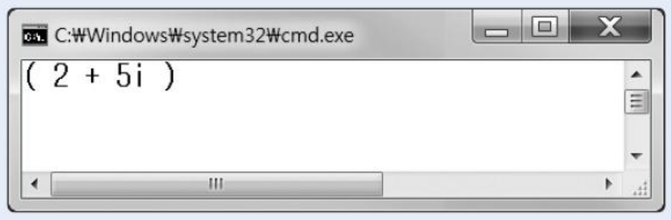
자동 인라인 함수

멤버함수의 정의가 아주 짧으면, 클래스 선언 내부에 함수를 직접 정의할 수도 있다. 그리고 클래스 내부에 정의된 함수는 함수 선언 앞에 inline이 없어도 자동으로 인라인 함수가 된다.

[예제 10-4]는 Complex 클래스의 멤버함수를 인라인 함수로 정의하는 프로그램이다. SetComplex 함수는 클래스 내부에 정의해서 자동 인라인 함수가 되도록 했고 ShowComplex 함수는 앞에 inline를 붙여 인라인 함수로 정의했다.

예제 10-4 인라인 함수 사용하기(10_04.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     void SetComplex()
10     {
11       real=2;
12       image=5;
13     }
14     void ShowComplex();
15 };
16
17
18 inline void Complex::ShowComplex()
19 {
```



```
20   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
21 }
22 void main()
23 {
24   Complex x;
25
26   x.SetComplex();
27   x.ShowComplex();
28 }
```

09행~13행 클래스 선언부에 멤버함수를 정의하면 자동 인라인 함수가 된다.

18행 예약어 inline을 사용해서 인라인 함수를 정의했다.

6_ const 상수와 const 멤버함수

#define문으로 정의한 매크로 상수보다 상수를 정의하는 더 쉬운 방법은 예약어 const를 사용하는 것이다. 일반 변수 선언과 유사하지만 반드시 초깃값을 주어야 한다.

const 상수 기본 형식

```
const 자료형 변수명 = 초깃값;
```

형식은 변수 선언과 같지만 키워드 const 다음의 변수명은 변수로서의 역할을 못한다. ❶과 같이 변수 선언 시 초깃값으로 지정한 값이 영원히 그 변숫값이 되어 상수로 사용된다. ❷는 잘못된 예다. const 상수는 반드시 초깃값을 주어야 하고 이미 선언이 끝난 const 상수에 대입 연산자로 값을 지정할 수 없다.

❶ `const double PI=3.141592; // 원주율`

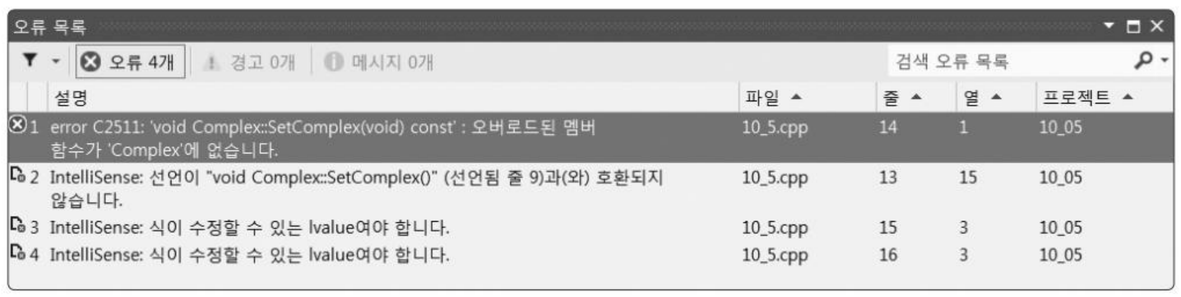
❷ `const double PI; // 에러`

const 상수는 매크로 상수와 동일한 목적으로 사용된다. 하지만 매크로 상수는 자료형을 명시하지 않지만 const 상수는 자료형을 명시한다는 점이 다르다.

const로 변수를 상수화하려면 초깃값을 주어야 한다.

클래스의 멤버함수를 정의할 때 const를 선언해서 멤버함수 내에서 어떤 멤버변수 값도 변경하지 못하도록 할 수 있다. Set으로 새롭게 값을 설정하는 멤버함수는 매개변수로 넘어 온 값으로 멤버변수 값을 변경해야 한다. 그러므로 SetXXX 함수들은 const를 붙일 수 없다. 만약 const를 붙이면 컴파일 에러가 발생하는데, 다음이 그 예다.

```
void Complex::SetComplex() const
{
  real=2;
  image=5;
}
```



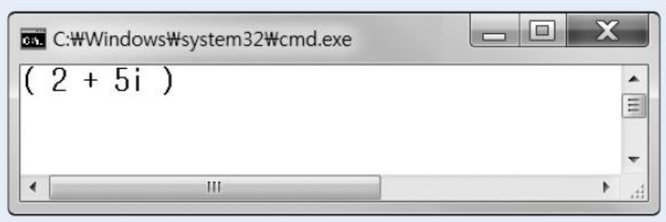
반면 GetXXX 함수들은 멤버변수 값을 알려주기 위한 용도로 사용된다. GetXXX 함수 내부에서 멤버변수 값을 변경하지 말아야 하므로 다음 예처럼 const 예약어로 함수를 정의할 때 기술한다.

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

[예제 10-5]는 Complex 객체에 값을 설정하는 ShowComplex 함수를 const 함수로 만드는 프로그램이다.

예제 10-5 const 멤버함수 사용하기(10_05.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
```



```
09     void SetComplex();
10     void ShowComplex() const;
11  };
12
13  void Complex::SetComplex()
14  {
15    real=2;
16    image=5;
17  }
18
19  void Complex::ShowComplex() const
20  {
21    cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
22  }
23
24  void main()
25  {
26    Complex x;
27    x.SetComplex();
28    x.ShowComplex();
29  }
```

09행, 13행~17행 SetComplex 멤버함수는 멤버변수 값을 변경하므로 const로 지정할 수 없다.

10행, 19행~22행 ShowComplex 함수 내부에서 멤버변수 값이 변경되지 말아야 하므로 예약어 const를 함수 뒤에 기술해서 const 함수로 정의한다.

7_ 함수의 오버로딩

C++의 특징에서 살펴봤듯이 C++에서는 다음과 같이 동일한 이름으로 함수를 여러 번 정의해서 사용할 수 있다.

```
void print(int);
void print(long);
void print(int, long);
```

그렇기 때문에 각 함수를 구분하기 위한 것이 필요한데, 이것이 바로 다음에 알아볼 '시그너처'다.

함수의 시그너처

함수 여러 개 중에서 특정 함수가 호출되면 컴파일러가 특정 함수를 구분해서 수행할 수 있어야 한다. 그러려면 프로그래머가 구분이 가능하도록 함수를 정의해야 한다. 컴파일러는 함수를 구분하기 위한 중요한 정보로 다음 3가지를 생각하는데, 이와 같이 함수를 구분하기 위한 구성요소를 시그너처(signature)라 한다.

❶ 함수명

❷ 매개변수의 개수

❸ 매개변수의 자료형

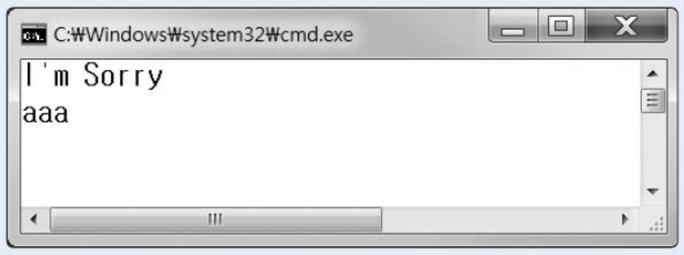
결국, 함수의 오버로딩은 함수명을 동일하게 해서 여러 번 오버로딩하되 함수를 호출할 때 모호하지 않도록 하려고 매개변수의 개수나 매개변수의 자료형을 다르게 주는 것을 의미한다. 이렇게 하면 함수를 호출할 때 이름이 같더라도 매개변수로 어떤 함수가 호출되었는지 구분할 수 있기 때문이다.

함수의 다형성(polymorphism)은 객체지향 프로그램의 특징 중 하나다. poly는 많다는 뜻이고 morphism은 형(형태)을 의미한다. 이를 우리말로 간단히 표현한 것이 다형성이다. 함수의 오버로딩도 다형성의 한 형태다. 왜냐하면 함수를 호출할 때에는 동일한 접근 방식으로 호출하지만 다양한 결과를 얻을 수 있기 때문이다.

[예제 10-6]은 함수의 오버로딩을 이용해서 문자열을 출력하는 함수를 작성하는 프로그램이다.

예제 10-6 함수의 오버로딩 살펴보기(10_06.cpp)

```
01 #include <iostream>
02 using namespace std;
03 void printstr(char *);
04 void printstr(char, int);
05
06 void main()
07 {
08   printstr("I'm Sorry");
09   printstr('a',3);
10 }
11
```



```
12 void printstr(char *the_string)
13 {
14   cout<<the_string<<endl;
15 }
16
17 void printstr(char the_char, int repeat_cnt)
18 {
19   for(int i = 0; i < repeat_cnt; i++)
20     cout<<the_char;
21   cout<<endl;
22 }
```

03행 문자열을 매개변수로 받는 printstr 함수의 선언부다.

04행 단일 문자와 정수를 매개변수로 받는 printstr 함수의 선언부다.

08행 큰따옴표로 묶은 문자열을 매개변수로 주었으므로 03행의 문자열을 매개변수로 받는 printstr 함수가 호출된다.

09행 작은따옴표로 묶은 문자와 정수가 주어졌으므로 04행의 단일 문자와 정수를 매개변수로 받는 printstr 함수가 호출되어 aaa가 출력된다.

12행~15행 문자열을 출력시키는 printstr 함수를 정의한다.

17행~22행 0부터 매개변수 repeat_cnt만큼 반복하면서 단일 문자인 매개변수 the_char를 출력하는 printstr 함수를 정의한다.

함수의 오버로딩이 필요한 이유

같은 의미로 사용하는 함수를 모두 다른 이름으로 정의한다면 프로그램을 작성할 때마다 함수명을 개별적으로 외워야 할 것이다. 예를 들어, 절댓값을 구하는 함수는 표준 함수로 다음과 같이 정의되어 있다.

- int abs(int x);
- double fabs(double x);
- long int labs(long int x);

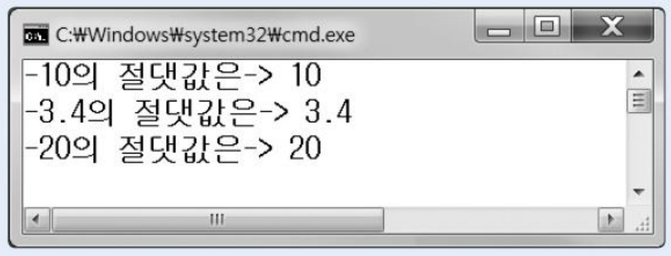
'절댓값 구하기'라는 동일한 목적을 수행하기 위해 우리가 기억할 함수는 3개나 된다(실제로는 이보다 더 많다). 만일 정수값에 대해서 절댓값을 구하려면 abs 함수를 사용해야 하고 fabs 함수는 실수형의 절댓값을 구할 때 사용한다. 또 labs 함수는 정수 중에서도 long int형의 절댓값을 구할 때 사용한다. 함수는 굉장히 많은데, 거기다 동일한 작업을 하려고

외워야 하는 함수가 또 이렇게 많다면 프로그램을 작성하는 작업이 더욱 어려울 것이다. 이럴 때 오버로딩을 통해 동일한 목적을 수행하기 위해 사용하는 함수명을 동일하게 사용할 수 있으므로 프로그램을 훨씬 쉽게 작성할 수 있다는 것이다.

[예제 10-7]은 함수를 구분하기 위한 요소로 함수명을 중요하게 취급한 경우로서, 절댓값을 구하는 함수를 서로 다른 이름으로 정의하는 프로그램이다.

예제 10-7 함수의 오버로딩 없이 절댓값 구하기(10_07.cpp)

```
01 #include <iostream>
02 using namespace std;
03 int myabs(int num)
04 {
05   if(num<0)
06     num=-num;
07   return num;
08 }
09
10 double fmyabs(double num)
11 {
12   if(num<0)
13     num=-num;
14   return num;
15 }
16
17 long int lmyabs(long int num)
18 {
19   if(num<0)
20     num=-num;
21   return num;
22 }
23
24 void main()
25 {
26   int a=-10;
27   cout << a <<"의 절댓값은-> "<< myabs(a) << endl;
28
29   double b=-3.4;
30   cout << b <<"의 절댓값은-> "<< fmyabs(b) << endl;
```



```
31
32    long int c=-20L;
33    cout << c <<"의 절댓값은-> "<< lmyabs(c) << endl;
34  }
```

03행~08행 int형 데이터의 절댓값을 구하는 함수 정의다.

10행~15행 double형 데이터의 절댓값을 구하는 함수 정의다.

17행~22행 long int형 데이터에 대해 절댓값을 구하는 함수 정의다.

27행 함수명이 abs이므로 03행~08행에 정의한 int형 데이터의 절댓값을 구하는 함수가 호출된다.

30행 함수명이 fabs이므로 10행~15행에 정의한 double형 데이터의 절댓값을 구하는 함수가 호출된다.

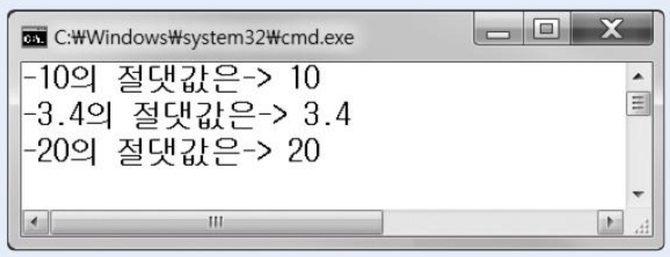
33행 함수명이 labs이므로 17행~22행에 정의한 long int형 데이터의 절댓값을 구하는 함수가 호출된다.

[예제 10-8]은 절댓값을 구하는 함수를 abs란 동일한 이름으로 3번 정의하되 매개변수의 자료형을 달리주어 함수를 구분할 수 있도록 한 프로그램이다. 단, [예제 10-7]과 비교해서 함수의 오버로딩을 사용했다는 점만 다르므로 결과 화면은 동일하다.

함수를 구분하기 위한 구성요소가 함수명 이외에도 매개변수가 있으므로 이 예제에서는 매개변수를 다르게 주어 정의할 것이다.

예제 10-8 매개변수의 자료형이 다른 함수의 오버로딩을 이용해서 절댓값 구하기(10_08.cpp)

```
01  #include <iostream>
02  using namespace std;
03  int abs(int num)
04  {
05    if(num<0)
06      num=-num;
07    return num;
08  }
09
10  double abs(double num)
11  {
12    if(num<0)
13      num=-num;
14    return num;
15  }
16
17  long int abs(long int num)
18  {
19    if(num<0)
```



```
20   num=-num;
21   return num;
22 }
23
24 void main()
25 {
26   int a=-10;
27   cout << a <<"의 절댓값은-> "<< abs(a) << endl;
28
29   double b=-3.4;
30   cout << b <<"의 절댓값은-> "<< abs(b) << endl;
31
32   long int c=-20L;
33   cout << c <<"의 절댓값은-> "<< abs(c) << endl;
34 }
```

03행~22행 함수를 동일한 이름으로 오버로딩하려면 다르게 지정해 주는 것이 있어야 한다. 그래야 호출될 때 어떤 함수가 호출되었는지 알 수 있는데, 여기서는 함수 3개에 사용된 매개변수 자료형이 다르다.

27행 함수를 호출할 때 int형 데이터를 실 매개변수로 주었으므로 03행~08행의 형식 매개변수를 int형으로 하는 abs 함수가 호출된다.

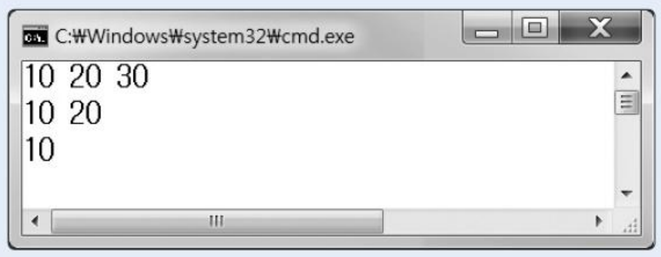
30행 double형 데이터를 실 매개변수로 주었으므로 10행~15행의 abs 함수가 호출된다.

33행 long int형 데이터를 실 매개변수로 주었으므로 17행~22행의 abs 함수가 호출된다.

[예제 10-8]에서 절댓값을 구하는 함수를 abs라는 동일한 이름으로 여러 번 정의해도 매개변수의 자료형이 다르면 해당 함수를 찾아갈 수 있었다. [예제 10-9]는 매개변수의 개수를 다르게 주는 함수의 오버로딩을 살펴보는 프로그램이다.

예제 10-9 매개변수의 개수가 다른 함수의 오버로딩 살펴보기(10_09.cpp)

```
01 #include <iostream>
02 using namespace std;
03 void print(int x, int y, int z)
04 {
05   cout<<x<<" "<<y<<" "<<z<<endl;
06 }
07 void print(int x, int y)
08 {
09   cout<<x<<" "<<y<<endl;
10 }
```



```
11 void print(int x)
12 {
13   cout<<x<<endl;
14 }
15 void main()
16 {
17   print(10, 20, 30);
18   print(10, 20);
19   print(10);
20 }
```

03행~06행 정수형 데이터 3개를 형식 매개변수로 하는 print 함수를 정의한다.

07행~10행 정수형 데이터 2개를 형식 매개변수로 하는 print 함수를 정의한다.

11행~14행 정수형 데이터 1개를 형식 매개변수로 하는 print 함수를 정의한다.

17행 정수형 데이터 3개를 실 매개변수로 지정해서 함수를 호출하면 03행~06행에 정의한 매개변수 abs 함수 3개가 호출된다.

18행 정수형 데이터 2개를 실 매개변수로 지정해서 함수를 호출하면 07행~10행에서 정의한 매개변수 abs 함수 2개가 호출된다.

19행 정수형 데이터를 1개를 실 매개변수로 지정해서 함수를 호출하면 11행~14행에서 정의한 매개변수 abs 함수 1개가 호출된다.

8. 함수의 기본 매개변수

C++ 함수만의 또 다른 특징을 살펴보자. 함수를 호출할 때는 함수 정의 시 기술한 매개변수의 개수를 반드시 맞추어서 호출해야 하지만 기본 매개변수를 사용하면 다르게 호출할 수 있다. 다음 예와 같이 함수의 형식 매개변수에 값을 설정할 수 있는데, 이렇게 형식 매개변수에 값을 설정해 놓은 것을 기본 매개변수라 한다.

```
void print(int x=10, int y=20, int z=30)
{
   cout<<x<<" "<<y<<" "<<z<<endl;
}
```

매개변수에 '기본(default)'이라는 용어를 붙인 이유는 기본값을 설정해 놓은 매개변수라는 의미를 주기 위해서다. 기본 매개변수는 함수를 호출할 때 대응되는 실 매개변수를 모두 기술하지 않아도 자동으로 적용되어 사용할 수 있는 매개변수다. 기본 매개변수를 사

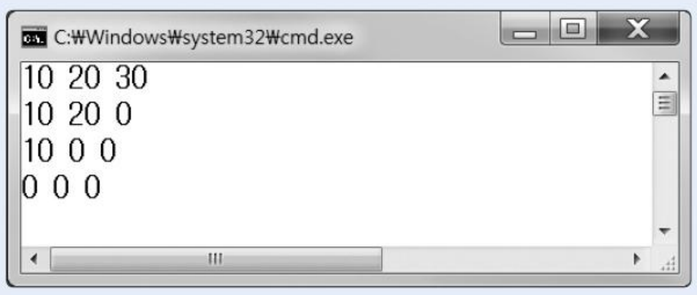
용할 때 주의할 점은 함수의 정의와 함수의 선언을 따로 할 경우, 기본 매개변수를 함수의 선언부에서 지정해줘야 한다는 점이다. 만약 함수의 선언 없이 함수를 main 함수 위에 정의했다면 기본 매개변수를 함수의 정의에 설정한다. 기본 매개변수를 갖는 함수는 다음과 같이 다양한 형태로 호출할 수 있다.

```
prn(4, 5, 6);
prn(4, 5);
prn(4);
prn();
```

[예제 10-10]은 함수의 선언에서 형식 매개변수에 기본값을 지정해서 함수를 다양하게 호출해보기 위한 프로그램이다.

예제 10-10 함수의 매개변수에 기본값 지정하기(10_10.cpp)

```
01 #include <iostream>
02 using namespace std;
03 void print(int x=0, int y=0, int z=0);
04 void main()
05 {
06   print(10, 20, 30);
07   print(10, 20);
08   print(10);
09   print();
10 }
11 void print(int x, int y, int z)
12 {
13   cout<<x<<" "<<y<<" "<<z<<endl;
14 }
```



03행 함수의 선언부에서 기본 매개변수를 지정한다.

06행 함수를 호출할 때 실 매개변수를 3개 모두 호출하면 기본 매개변수에 지정해 준 값이 적용되지 않는다. 그래서 출력 결과는 10 20 30이다.

07행 실 매개변수를 2개만 호출하면 앞에 기술된 매개변수부터 값이 전달되므로 x에는 10이 y에는 20이 저장되고, c에는 값을 주지 않았기 때문에 기본 매개변수에 설정된 값인 0이 적용된다. 출력 결과는 10 20 0이다.

08행 실 매개변수를 1개만 호출하면 10이 x에 저장되고 y와 z에는 전달해 준 값이 없으므로 각 0과 0이 저장된다. 출력 결과는 10 0 0이다.

09행 실 매개변수를 주지 않으면 기본 매개변수 값이 모두 적용되어서 출력 결과는 0 0 0이다.

Section

02 생성자와 소멸자

1_ 생성자의 의미와 특징

기본 자료형으로 변수를 선언할 때 선언과 동시에 값을 주는 것을 초기화라 한다. 클래스도 객체를 생성할 때 기본 자료형처럼 초깃값을 줄 수 있어야 하는데, 이를 가능하게 하는 것이 생성자다.

다음은 기본 자료형인 int형으로 변수를 정의하는 방법이다. 두 문장이 의미하는 것이 무엇인지 비교해 보자.

❶
```
int a;  // 변수의 선언
a=5;    // 변수에 값을 대입
```

❷
```
int a=5;  // 선언과 동시에 초기화
```

❶은 변수를 먼저 선언한 후에 대입 연산자로 변수에 값을 대입한다. 반면, ❷는 정수형 변수를 선언함과 동시에 값을 지정해서 초기화한다. 이처럼 초기화는 변수가 메모리 할당을 진행하는 동시에 값을 설정하는 것으로, 변수를 선언할 때 한 번 밖에 사용하지 못한다. 반면, 이미 선언된 변수에 대입 연산자로 값을 지정하는 것은 어디서나 몇 번이고 사용할 수 있다. 변수의 초기화가 변수 선언 시 값을 설정하는 것처럼 객체의 초기화도 객체를 선언할 때 값을 설정한다. 그런데 객체의 초기화는 생성자라는 멤버함수가 필요하다. 이 생성자가 객체를 생성할 때 자동으로 호출되어 변수의 초기화처럼 객체를 초기화해 준다.

생성자의 특징

다음은 생성자의 특징이자 생성자가 있어야 하는 이유다.

❶ 생성자는 특별한 멤버함수다.

❷ 생성자명은 클래스명과 동일하다.

❸ 생성자는 자료형(반환값의 유형)을 지정하지 않는다.

❹ 생성자의 호출은 명시적이지 않다.

❺ 생성자는 객체를 선언(생성)할 때 컴파일러에 의해 자동으로 호출된다.

❻ 객체의 초기화란 멤버변수의 초기화를 의미한다.

프로그래머가 생성자를 명시적으로 만들지 않으면 C++ 컴파일러는 매개변수가 없는 생성자를 자동으로 만들어 놓는다. 컴파일러가 만든 매개변수가 없는 생성자는 아무런 일도 하지 않는데, 이러한 생성자를 '기본 생성자(default constructor)'라고 한다. 예를 들어, 다음과 같이 기술하면 기본 생성자를 호출하는 것이 된다.

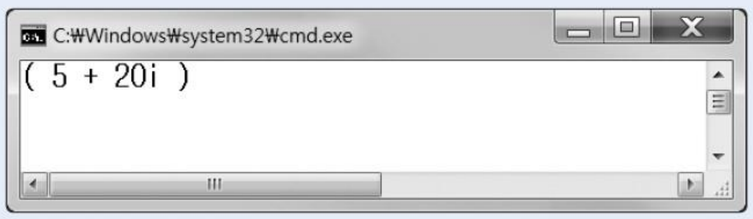
```
Complex x;
```

컴파일러에서 자동으로 주는 생성자를 사용하면 아무 일도 하지 않기 때문에 멤버변수에 쓰레기 값이 저장된다. 그러므로 객체가 생성될 때 멤버변수에 특정한 값을 저장하려면 프로그래머가 매개변수가 없는 생성자를 재정의해 주어야 한다.

[예제 10-11]은 매개변수가 없는 기본 생성자를 C++ 컴파일러가 제공해 주는 것과 별개로 명시적으로 생성자를 정의하고 호출하는 프로그램이다.

예제 10-11 매개변수가 없는 생성자 작성하기(10_11.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     Complex();
10     void ShowComplex() const;
11 };
12
13 Complex::Complex()
14 {
15   real=5;
16   image=20;
17 }
18
19 void Complex::ShowComplex() const
```



```
20 {
21   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
22 }
23
24 void main()
25 {
26   Complex x;
27   x.ShowComplex();
28 }
```

09행 생성자명은 클래스명과 같다. 생성자는 멤버함수이므로 반드시 클래스 내부에 선언하고 정의한 후에 호출되어야 한다. 생성자의 선언을 보면 함수명이 클래스명인 Complex와 동일한 것을 확인할 수 있다. 또 주의해서 봐야 할 것은 함수의 자료형이 없다는 것이다.

```
자료형 함수명(매개변수 리스트);
```

일반 함수의 선언문은 위와 같지만 생성자는 함수의 자료형을 생략한다. 그리고 여기서는 기본 생성자에 대한 선언이므로 매개변수도 없다.

13행~17행 09행처럼 생성자를 정의한다. 역시 함수의 반환값에 대한 자료형이 없고 매개변수도 없다. 생성자 정의를 클래스 외부에서 하기 때문에 생성자 앞에 소속 클래스명을 스코프 연산자와 함께 기술했다. 생성자는 객체를 초기화해야 한다고 했는데, 객체의 초기화란 결국 객체를 구성하는 멤버변수를 초기화하는 것이므로 생성자에서는 클래스 Complex의 멤버변수인 real과 image를 초기화했다.

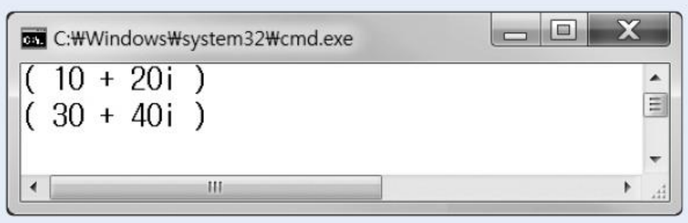
26행 객체를 생성하는 객체 선언문이다. 아무리 봐도 생성자 함수를 호출하고 있는 것처럼 보이지 않는다. 개체를 선언하면 생성자가 자동으로 호출되기 때문이다.

27행 26행에서 객체 생성 시 생성자가 자동 호출되었다는 것은 객체 x를 출력한 결과를 보면 알 수 있다. 객체를 선언한 후 곧바로 출력했는데, (5 + 20i)가 출력되었다. 생성자가 호출되지 않았다면 쓰레기 값이 출력되었을 텐데 프로그래머가 작성한 생성자에서 각 멤버변수에 초깃값으로 준 값이 그대로 출력되었으므로 생성자가 호출된 것을 확인할 수 있다.

앞서 정의한 매개변수가 없는 기본 생성자는 모든 객체에 대해서 항상 초깃값이 동일하다. 이제 객체마다 다양한 초깃값의 매개변수를 갖는 생성자를 만들어 보자.

예제 10-12 다양한 초깃값의 매개변수를 사용하는 생성자 작성하기(10_12.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
```



```
06     int real;
07     int image;
08   public :
09     Complex(int r, int i);
10     void ShowComplex() const;
11 };
12
13 Complex::Complex(int r, int i)
14 {
15   real=r;
16   image=i;
17 }
18
19 void Complex::ShowComplex() const
20 {
21   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
22 }
23
24 void main()
25 {
26   Complex x(10, 20);
27   Complex y(30, 40);
28   // Complex z;
29   x.ShowComplex();
30   y.ShowComplex();
31 }
```

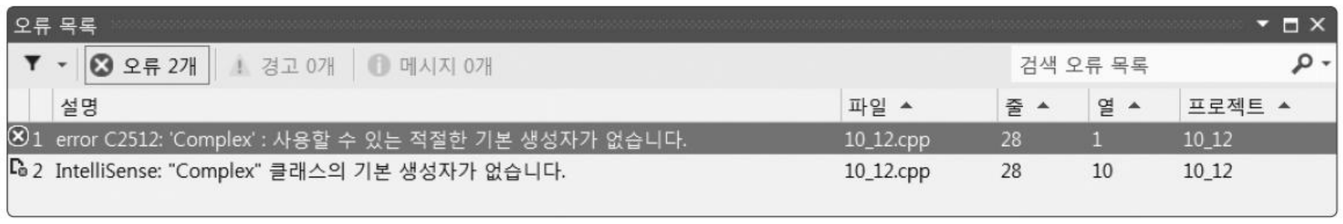
09행 매개변수 2개를 갖는 생성자 선언이다.

13행~17행 생성자는 매개변수 2개 중 첫 번째 매개변수인 r은 실수부(real)에, 두 번째 매개변수인 i는 허수부(image)에 저장한다. 생성자 함수는 일반 멤버함수의 정의와 거의 유사하다. 다른 점은 함수의 자료형이 없다는 것과 함수명이 반드시 클래스명과 일치해야 한다는 것이다. 또한 생성자가 멤버함수와 가장 큰 차이점은 명시적으로 호출되지 않고 객체 선언 시 자동으로 호출된다는 것이다.

26행~27행 객체를 선언할 때 객체명 다음에 소괄호가 나타난다. 소괄호는 생성자에 전달해 줄 실 매개변수를 기술한다. 이렇게 하면 객체 초기화를 위해 만들어 준 생성자가 자동 호출된다. 생성자는 암시적으로 호출되지만 매개변수는 클래스 외부에서 주어야 하므로 객체명 다음에 소괄호를 기술해서 값을 설정해 주어야 한다.

2. 생성자 오버로딩

객체를 선언할 때도 초깃값을 주지 않을 수 있다. 매개변수가 없는 기본 생성자는 C++ 컴파일러가 제공해 준다. 그런데 프로그래머가 매개변수를 갖는 기본 생성자를 만들어 주면 컴파일러는 더 이상 기본 생성자를 제공하지 않고 프로그래머가 직접 매개변수 없는 생성자를 작성할 것을 떠맡긴다. 이러한 기본 생성자의 특징을 증명하기 위해 [예제 10-12]에서 28행의 주석문을 제거하면 다음과 같은 에러가 발생한다.

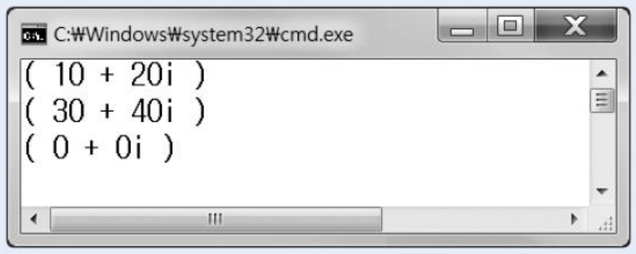


[예제 10-12]의 28행에서 발생한 컴파일 에러를 없애려면 기본 생성자를 하나 더 추가해서 정의해야 한다. 즉, 매개변수가 2개인 생성자를 만들어 사용하던 중 매개변수를 갖지 않는 생성자가 필요하다면 이를 프로그래머가 직접 정의하고 사용해야 한다는 결론을 내릴 수 있다. 동일한 이름의 함수를 여러 번 정의할 수 있는 함수의 오버로딩은 생성자에도 적용된다. 생성자도 함수의 일종이므로 매개변수의 개수나 자료형을 달리해서 여러 번 정의할 수 있는데, 이를 생성자 오버로딩이라고 한다.

[예제 10-13]은 생성자 오버로딩을 이용해서 다양한 형태로 객체를 생성하는 프로그램이다.

예제 10-13 생성자 오버로딩하기(10_13.cpp)

```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     Complex();
10     Complex(int r, int i);
11     void ShowComplex() const;
```



```
12 };
13 Complex::Complex()
14 {
15   real=0;
16   image=0;
17 }
18
19 Complex::Complex(int r, int i)
20 {
21   real=r;
22   image=i;
23 }
24
25 void Complex::ShowComplex() const
26 {
27   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
28 }
29
30 void main()
31 {
32   Complex x(10, 20);
33   Complex y(30, 40);
34   Complex z;
35   x.ShowComplex();
36   y.ShowComplex();
37   z.ShowComplex();
38 }
```

09행 기본 생성자를 선언했다.

10행 생성자가 매개변수 2개를 갖는다. 09행에 이미 생성자가 있지만 매개변수의 자료형과 개수를 달리 주어 생성자를 한 번 더 선언했다. 이를 생성자 오버로딩이라고 한다.

13행~17행 외부로부터 아무런 값도 전달받지 않는 매개변수가 없는 기본 생성자를 정의했다.

19행~23행 외부로부터 값 2개를 전달받는 매개변수 생성자 2개를 정의한다.

32행~33행 매개변수가 2개인 생성자를 호출한다. 객체를 생성할 때 외부에서 준 값으로 초기화한다.

34행 객체를 생성할 때 초깃값을 지정하지 않았기 때문에 기본 생성자가 호출된다.

3_ 생성자의 기본 매개변수 값 지정하기

객체의 초깃값을 다양하게 주기 위한 방법의 하나로 생성자의 오버로딩을 살펴봤다. 하지만 이보다 더 편리한 방법이 있다. 바로 생성자에 기본 매개변수 값을 설정하는 것인데, 다음 예처럼 함수 선언문의 매개변수에 대입 연산자를 사용해서 기본 매개변수 값을 함께 주는 것이다.

```
Complex(int r=0 , int i=0);
```

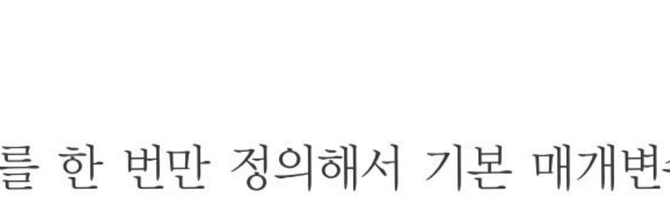
함수의 선언문에 기본 매개변수 값을 기술하면 함수를 한 번만 정의해서 기본 매개변수 값을 다양한 형태로 호출할 수 있다. 기본 매개변수 값을 지정한 멤버함수는 실 매개변수 없이 다음과 같이 실 매개변수 1개, 실 매개변수 2개 등으로 다양하게 호출할 수 있다는 장점이 있다.

```
Complex x(10, 20);  // 실 매개변수 2개를 설정해서 호출
Complex y(30);      // 실 매개변수 1개만을 설정해서 호출
Complex z;          // 매개변수 없이 호출
```

[예제 10-14]는 생성자에 기본 매개변수 값을 설정해서 객체를 다양하게 초기화하는 프로그램이다.

예제 10-14 생성자의 기본 매개변수 값 설정하기(10_14.cpp)

```
01  #include <iostream>
02  using namespace std;
03  class Complex
04  {
05    private :
06      int real;
07      int image;
08    public :
09      Complex(int r=0, int i=0);
10      void ShowComplex() const;
11  };
12
13  Complex::Complex(int r, int i)
14  {
15    real=r;
```



```
16   image=i;
17 }
18
19 void Complex::ShowComplex() const
20 {
21   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
22 }
23
24 void main()
25 {
26   Complex x(10, 20);
27   Complex y(30);
28   Complex z;
29   x.ShowComplex();
30   y.ShowComplex();
31   z.ShowComplex();
32 }
```

13행~17행 생성자를 하나만 정의한다. 하지만 26행~28행에서 호출된 형태는 3가지다.

09행 생성자를 하나만 정의했으나 다양한 형태로 호출할 수 있는 이유는 생성자에 기본 매개변수 값을 지정해 주었기 때문이다. 기본 매개변수 값은 함수의 선언부에서 설정해 주어야 한다. 생성자를 선언하는 부분에서 실수부(r)에 0을, 허수부(i)에도 0을 기본 매개변수 값으로 설정했다.

26행 객체를 초기화할 때 매개변수를 10과 20으로 2개를 지정하면 r에는 10이 i에는 20이 저장된다. 즉, 기본 매개변수 값이 무시된다.

27행 객체를 초기화할 때 매개변수를 한 개만 30으로 지정하면 그 지정한 값 30이 r에 저장되고, i는 초깃값을 명시하지 않았으므로 기본 매개변수 값이 적용되어 0이 저장된다.

28행 객체를 초기화할 때 값을 주지 않으면 기본 매개변수 값이 2개 모두 적용된다. 그래서 r과 i 모두에 0이 저장된다.

4. 생성자의 콜론 초기화

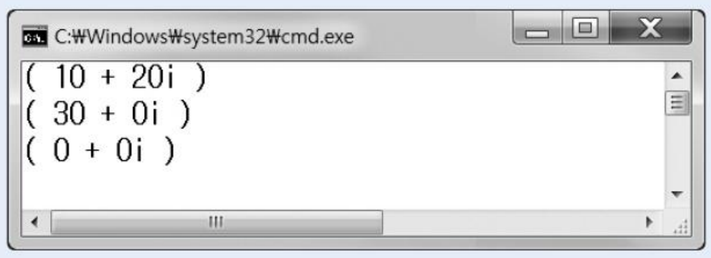
생성자는 객체를 초기화하기 위한 멤버함수다. 그리고 객체의 초기화는 결국 멤버변수에 초깃값을 설정하는 것이다. 그러므로 생성자에서 주로 일어나는 일은 멤버변수에 대입 연산자를 사용해서 값을 설정하는 것이다. 일반적으로 멤버변수에 초깃값을 설정하는 작업은 생성자 함수의 몸체에 기술한다. 하지만 생성자의 머리 부분에서 멤버변수에 초깃값을 설정할 수도 있는데, 이때 다음과 같이 콜론 초기화가 사용된다.

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

함수의 몸체에 기술되어야 할 멤버변수의 초깃값 설정을 함수의 머리 부분으로 옮기되 콜론으로 멤버변수를 초기화한다고 알린다. 콜론 다음에는 초깃값을 설정할 멤버변수 다음에 소괄호를 기술한 후 소괄호 안에 초깃값을 기술한다. 초기화할 변수가 여러 개이면 콤마로 연결해서 반복적으로 기술하면 된다.

다음은 생성자의 함수 몸체에서 멤버변수에 초깃값을 설정하는 것을 함수의 머리 부분에서 콜론을 초기화하는 것으로 변경한 프로그램이다. 결과는 [예제 10-14]와 동일하다.

예제 10-15 생성자 콜론 초기화하기(10_15.cpp)




```
01 #include <iostream>
02 using namespace std;
03 class Complex
04 {
05   private :
06     int real;
07     int image;
08   public :
09     Complex(int r=0, int i=0);
10     void ShowComplex() const;
11 };
12
13 Complex::Complex(int r, int i) : real(r), image(i)
14 {
15 }
16
17 void Complex::ShowComplex() const
18 {
19   cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
20 }
21
22 void main()
23 {
24   Complex x(10, 20);
25   Complex y(30);
```

```
26   Complex z;
27   x.ShowComplex();
28   y.ShowComplex();
29   z.ShowComplex();
30 }
```

13행~15행 생성자로 전달된 매개변수 값을 멤버변수로 초기화하는 문장이다. 함수의 몸체에 기술하는 멤버변수의 초기화를 대신할 수 있는 문장이 바로 13행의 콜론 초기화다.

5. 소멸자의 의미와 특징

생성자와 함께 다루어야 할 멤버함수가 소멸자(destructor)다. 소멸자는 생성자의 반대되는 개념으로 소멸자에 대한 이해를 도우려고 소멸자를 생성자와 비교해 보면 다음과 같다.

❶ 생성자(constructor)는 객체가 생성될 때 자동 호출되고 소멸자(destructor)는 객체가 소멸될 때 자동으로 호출된다.

❷ 생성자가 객체를 초기화하기 위한 멤버함수라면 소멸자는 객체를 정리해 주는(리소스를 해제한다든지 하는 작업) 멤버함수다.

소멸자의 특징

소멸자도 생성자처럼 매개변수가 없는 기본 생성자를 C++ 컴파일러로부터 자동으로 제공받는다. 그러므로 모든 클래스에 명시적으로 기술하지 않아도 된다. 그런데 C++ 컴파일러가 제공하는 소멸자는 아무 일도 하지 않기 때문에 객체가 소멸될 때 해야 할 일이 있다면 프로그래머가 직접 명시해 주어야 한다. 이러한 소멸자는 다음과 같은 특징이 있다.

❶ 소멸자 함수는 멤버함수다.

❷ 소멸자 함수명도 생성자처럼 클래스명을 사용한다.

❸ 소멸자 함수는 생성자 함수와 구분하려고 함수명 앞에 ~ 기호를 붙인다.

❹ 소멸자는 자료형을 지정하지 않는다(반환값의 유형을 지정하지 않는다).

❺ 소멸자의 호출은 명시적이지 않다.

❻ 소멸자는 객체 소멸 시 자동 호출된다.

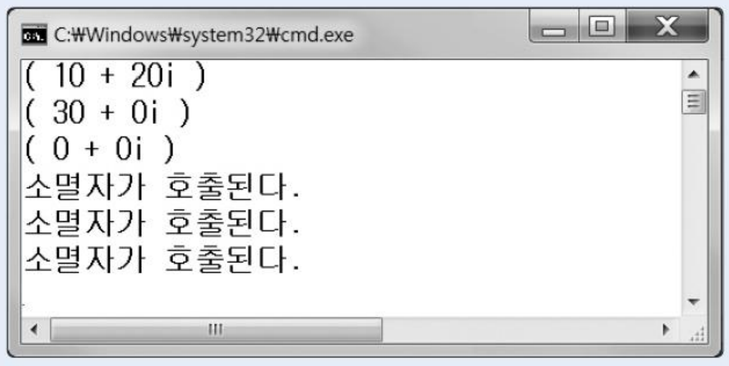
❼ 소멸자는 전달 매개변수를 지정할 수 없다.

❽ 소멸자는 전달 매개변수를 지정할 수 없으므로 오버로딩 할 수 없다.

[예제 10-16]은 클래스에 소멸자를 추가하고 소멸자가 어느 시점에서 자동으로 호출되는지를 살펴보는 프로그램이다.

예제 10-16 소멸자 정의하기(10_16.cpp)

```
01  #include <iostream>
02  using namespace std;
03  class Complex
04  {
05    private :
06      int real;
07      int image;
08    public :
09      Complex(int r=0, int i=0);
10      ~Complex();
11      void ShowComplex() const;
12  };
13
14  Complex::Complex(int r, int i) : real(r), image(i)
15  {
16  }
17
18  Complex::~Complex()
19  {
20    cout<<"소멸자가 호출된다. \n";
21  }
22
23  void Complex::ShowComplex() const
24  {
25    cout<<"( " <<real <<" + " <<image << "i )" <<endl ;
26  }
27
28  void main()
29  {
30    Complex x(10, 20);
31    Complex y(30);
32    Complex z;
33    x.ShowComplex();
```



```
34    y.ShowComplex();
35    z.ShowComplex();
36  }
```

10행 소멸자의 선언이다. 클래스명이 Complex이므로 소멸자의 이름도 Complex다. 하지만 생성자와 달리 소멸자는 이름 앞에 붙은 ~ 기호를 붙인다.

18행~21행 대부분의 클래스에는 소멸자를 정의하지 않고 C++ 컴파일러에서 제공해주는 기본 소멸자를 사용한다. 여기서는 소멸자가 특별한 역할을 하지 않지만 소멸자의 형태와 선언 방식을 살펴보기 위해 명시적으로 만들어 주었다.

소멸자명은 ~Complex이고 함수명 앞에 붙은 Complex::는 소속된 클래스명을 ::와 함께 기술한 것이다. 소멸자도 멤버함수의 일종이므로 소속된 클래스를 밝혀야 한다. 클래스명과 소멸자명은 모두 Complex다.

단지 소멸자는 앞에 ~ 기호가 붙는다. main 함수를 살펴보면 소멸자가 호출된 곳을 어디에서도 찾을 수 없다. 소멸자는 컴파일러에 의해 암시적으로 호출되기 때문이다. 소멸자가 C++ 컴파일러에 의해 호출되는지를 확인하려고 소멸자에 간단한 메시지가 출력되도록 구현했다.

저자 한마디

소멸자가 호출되는 시점

객체가 소멸될 때 컴파일러가 소멸자를 호출하기 때문에 소멸자가 호출되는 시점은 컴파일러가 결정한다. 그리고 객체가 소멸되는 시점은 기억 클래스와 연관이 있다. 기억 클래스에 의해 변수의 생존 기간이 결정되는데, 객체는 모든 면에서 변수와 성격이 유사하다. 객체도 기억 클래스에 의해 소멸되는 시점이 달라지고 이에 따라 소멸자가 호출되는 시점도 달라진다. 위의 예로 든 객체 x는 스택에 저장되는 자동변수이고 main 함수 내부의 객체 선언부에 의해 생성되었으므로 main 함수가 종료될 때 소멸된다.

✻ 요약 / 연습문제

[요약]

1 객체지향 프로그래밍은 프로그램을 작성하기 위해 필요한 객체들을 먼저 생각하고 필요한 함수를 사용하면서 프로그램을 작성해 나간다.

2 객체지향 프로그래밍은 캡슐화(encapsulation)와 데이터 은닉(data Hiding), 다형성(poly morphism), 함수의 오버로딩, 연산자 오버로딩(operator overloading), 상속성(inheritance)과 같은 특징이 있다.

3 클래스의 구조는 클래스 선언과 클래스 멤버함수의 정의로 나눌 수 있는데, 클래스 선언에는 멤버변수와 멤버함수의 원형 정의(prototype)를 기술하고 멤버함수의 정의는 클래스 선언 밖에서 따로 이루어진다.

4 접근 지정자는 클래스에서 각 멤버변수나 멤버함수 앞에 붙여서 사용하는 예약어인데, 클래스 내부에 선언된 멤버변수와 멤버함수의 접근 권한을 정한다.

5 프로그램 내에서 실질적인 작업을 수행하려면 클래스로 객체를 선언해야만 하는데, 객체는 클래스를 실체(instance)화한 것이다.

6 프로그램 속도를 높이기 위해 보강한 함수가 인라인 함수인데, 매크로 함수와 동작 원리가 동일하다. 클래스 내부에 멤버함수를 정의하면 자동 인라인 함수가 된다.

7 const 멤버함수는 함수 내부에서 멤버변수 값을 변경하지 못하도록 한다.

8 함수명을 동일하게 주고 여러 번 정의할 수 있는데, 이를 함수의 오버로딩이라고 한다. 함수의 오버로딩은 함수명을 동일하게 주는 대신 매개변수의 개수나 자료형을 달리 주어야 한다.

9 객체 초기화를 하기 위한 멤버함수로서 생성자가 있는데, 다음과 같은 특징이 있다.

❶ 생성자는 특별한 멤버함수다.
❷ 생성자명은 클래스명과 동일하다.
❸ 생성자의 자료형(반환값의 유형)을 지정하지 않는다.
❹ 생성자의 호출은 명시적이지 않는다.
❺ 생성자는 객체를 선언(생성)할 때 컴파일러에 의해 자동으로 호출된다.
❻ 객체의 초기화란 멤버변수의 초기화를 의미한다.

10 객체가 소멸될 때 자동 호출되는 소멸자가 있는데, 다음과 같은 특징이 있다.

❶ 소멸자는 멤버함수다.
❷ 소멸자의 이름 역시 생성자와 마찬가지로 클래스명을 사용한다.
❸ 소멸자는 생성자 함수와 구분하기 위해 함수명 앞에 ~ 기호를 덧붙인다.
❹ 소멸자의 자료형을 지정하지 않는다(반환값의 유형을 지정하지 않는다).
❺ 소멸자의 호출은 명시적이지 않다.
❻ 소멸자는 객체 소멸시 자동 호출된다.
❼ 소멸자는 전달 매개변수를 지정할 수 없다.
❽ 소멸자는 전달 매개변수를 지정할 수 없으므로 오버로딩할 수 없다.

[연습문제]

1 다음 중 생성자에 대한 설명으로 옳지 않은 것은?

❶ 생성자명은 클래스명과 동일하다.
❷ 생성자는 리턴형을 언급하지 않는다.
❸ 생성자는 객체가 생성될 때 자동으로 호출되는 함수다.
❹ 클래스에는 개발자가 반드시 직접 정의해야만 한다.

2 함수를 오버로딩하기 위해서 달라야 하는 함수의 구성요소가 아닌 것을 모두 고르시오.

❶ 반환형
❷ 매개변수의 개수
❸ 매개변수의 타입
❹ 함수명

3 다음 프로그램을 실행하면 에러가 발생한다. 에러가 발생하지 않도록 수정하시오.

```
01  #include <iostream>
02  using namespace std;
03  class CRect{
04      int left;
05      int top;
06      int right;
07      int bottom;
```

연습문제

```
08   public :
09     void print();
10  };
11  void CRect::print()
12  {
13    cout << "( " << left << ", " << top << ", " << right << ", " <<
14    bottom << " )" << endl ;
15  }
16  void main()
17  {
18    CRect rc;
19    rc.left=0;
20    rc.top=0;
21    rc.right=20;
22    rc.bottom='20';
23    rc.print();
24  }
```

4 SetRect()를 멤버함수로 정의하시오.

```
01  #include <iostream>
02  using namespace std;
03  class CRect{
04     int left;
05     int top;
06     int right;
07     int bottom;
08   public :
09     void print();
10     void SetRect(int l, int t, int r, int b);
11  };
12  void CRect::print()
13  {
14    cout << "( " << left << ", " << top << ", " << right << ", " <<
15    bottom << " )" << endl ;
16  }
17  void main()
```

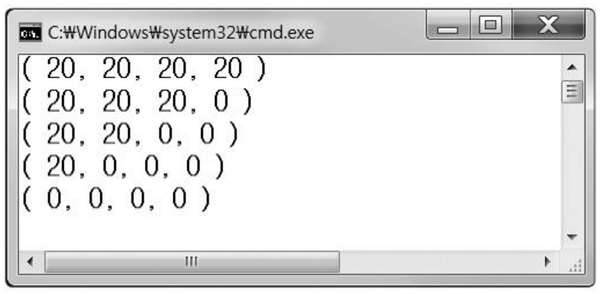
```
18 {
19   CRect rc;
20   rc.SetRect(0, 0, 20, 20); // error
21   rc.print();
22 }
```

5 객체를 초기화할 수 있도록 생성자를 오버로딩하시오.

```
01 #include <iostream>
02 using namespace std;
03 class CRect{
04    int left;
05    int top;
06    int right;
07    int bottom;
08 public :
09    void prn();
10 };
11 void CRect::prn()
12 {
13   cout << "( " << left << ", " << top << ", " << right << ", " <<
14   bottom << " )" << endl ;
15 }
16
17 void main()
18 {
19   CRect rc(0, 0, 20, 20);
20   CRect rc2(20, 20);
21   CRect rc3;
22   rc.prn();
23   rc2.prn();
24   rc3.prn();
25 }
```

연습문제

6 객체를 초기화할 수 있도록 생성자에 기본 매개변수 값으로 할당받도록 정의하시오.




```
01  #include <iostream.h>
02  #include <iomanip.h>
03  class CRect{
04     int left;
05     int top;
06     int right;
07     int bottom;
08    public :
09     void prn();
10  };
11
12  void CRect::prn()
13  {
14    cout << "( " << left << ", " << top << ", "
15         << right << ", " << bottom << " )" << endl ;
16  }
17
18  void main()
19  {
20    CRect rc( 20, 20, 20, 20 );
21    CRect rc2( 20 ,20, 20 );
22    CRect rc3( 20, 20 );
23    CRect rc4( 20 );
24    CRect rc5;
25
26    rc.prn();
27    rc2.prn();
28    rc3.prn();
29    rc4.prn();
30    rc5.prn();
31  }
```