



## Chapter 10. 클래스와 객체

# 목차

1. 객체지향 프로그래밍의 특징
2. 클래스의 이해
3. 생성자와 소멸자

# 학습목표

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

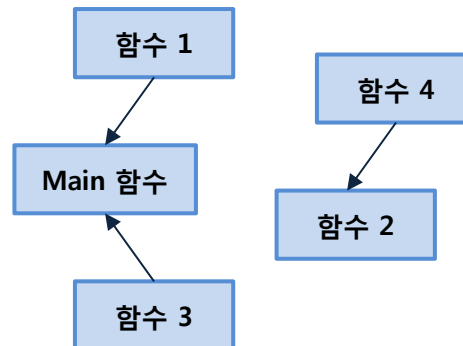
# 00. 객체지향 프로그래밍의 특징

## ■ 객체지향 언어인 C++

- ① 절차적 프로그래밍 : 프로그램이 함수의 집합으로 구성되므로 함수를 정의하면서 함수에 필요한 데이터를 선언하여 사용한다. 대표적인 언어로는 C가 있다.
- ② 객체지향 프로그래밍 : 객체를 생산하기 위한 클래스를 설계한 후에 이를 다룰 함수(사용자 인터페이스)를 정의하여 함수로 객체를 다루도록 한다. 대표적인 언어로는 C++, 자바 등이 있다.

## ■ 절차적 프로그래밍

- 함수를 중심으로 프로그램을 설계한 후 함수에 필요한 데이터를 정의한다.

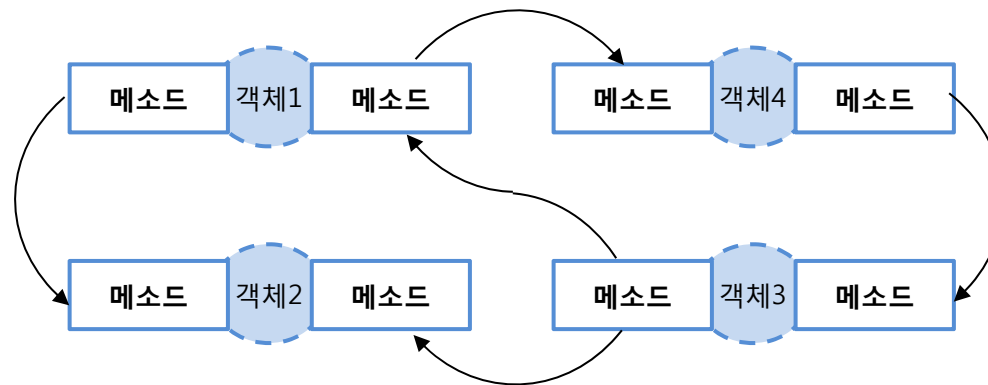


[그림 1-2] 절차적 프로그래밍

# 00. 객체지향 프로그래밍의 특징

## ■ 객체지향 프로그래밍

- 실 세계의 현상을 객체로 모델화 함으로써 다양한 문제를 해결하기 위한 프로그램 기법. 여기에서 객체는 실체(데이터)와 그 실체와 관련되는 동작(절차, 방법, 기능)을 모두 포함한다.
- 예시) 기차역에서 승차권을 발매하는 예를 들면, 실체인 '손님'과 절차인 '승차권 주문'은 하나의 객체이고, 실체인 '역무원'과 절차인 '승차권 발매'도 하나의 객체이다. 승차권 발매를 위해서는 객체 간에는 메시지 ("부산 가는 KTX 표한장 주세요")를 주고받는다. 메시지를 받은 객체는 동작(절차)을 실행한다.
- 객체 지향 프로그래밍은 시스템의 모듈화, 캡슐화를 촉진하여 복잡화, 거대화되는 소프트웨어를 사용하기 쉽고, 작성하기 쉬우며, 유지 보수하기 쉬운 방향으로 재구축하는 기법이다.



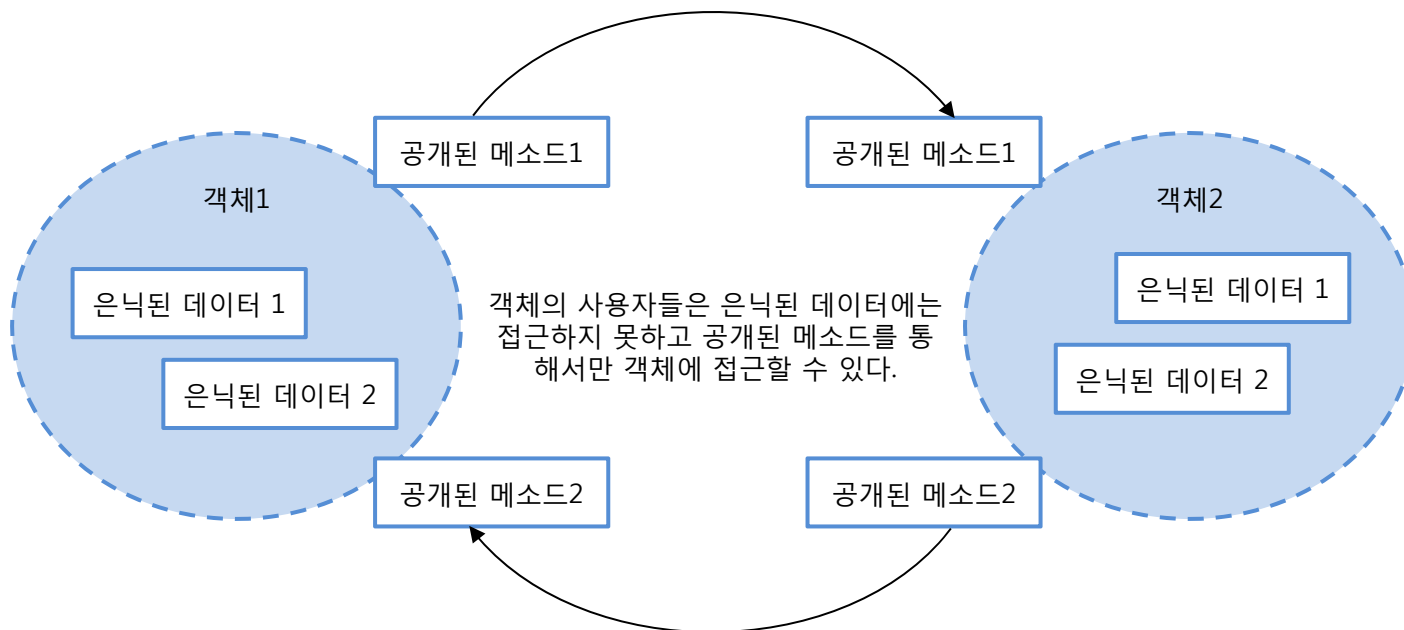
[그림 1-3] 객체지향 프로그래밍

# 00. 객체지향 프로그래밍의 특징

## ■ 객체지향 프로그래밍의 주요 특징

### ① 캡슐화(encapsulation)

- 데이터를 직접 다루면 데이터가 손상될 수 있으므로 이를 방지하기 위해 제공되는 것이 캡슐화(encapsulation)와 데이터 은닉(data hiding)이다.
- 즉, 객체의 상세한 내용을 객체 외부에 철저히 숨기고 단순히 메시지만으로 객체와의 상호작용을 하게 하는 것.



[그림 1-4] 객체의 은닉된 데이터와 공개된 함수

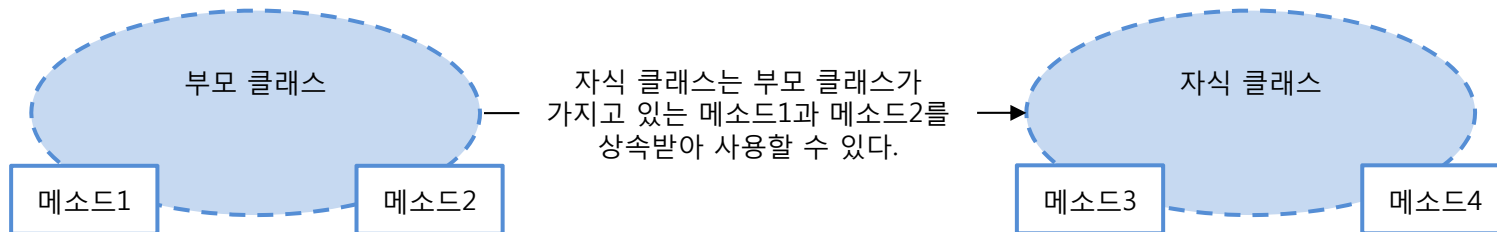
# 00. 객체지향 프로그래밍의 특징

## ② 다형성(polymorphism)

- 다형성은 하나의 인터페이스를 이용하여 서로 다른 구현방법을 제공하는 것.
- 오버로딩이나 오버라이딩을 통하여 동일한 이름의 함수나 연산자를 경우에 따라 다르게 동작할 수 있도록 함.

## ③ 상속성(inheritance)

- 상속성은 특정 객체의 성격(데이터와 메소드)을 다른 객체가 상속받아 사용할 수 있도록 하는 것.



# 01 클래스의 이해

## ■ 클래스의 선언

- C++에서 클래스를 선언하는 예약어는 **class**다. class는 자료를 추상화해서 사용자정의 자료형으로 구현할 수 있게 하는 C++의 도구다. 클래스는 크게 **클래스 선언**과 **클래스 멤버함수 정의**로 구성되어 있다.

클래스 선언	클래스 멤버함수 정의
<pre>class 클래스명 {   접근 지정자 :   자료형 멤버변수;    접근 지정자 :   자료형 멤버함수(); };</pre> <div> <div>멤버변수 선언</div> <div>멤버변수 원형 정의</div> </div>	<pre>자료형 클래스명::멤버함수() {   ...   ... }</pre>

[표 10-1] 접근 지정자

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

- private : 해당 멤버가 속한 클래스의 멤버함수에서만 사용 가능하며, 캡슐화(데이터 은닉)된다.
- public : 객체를 사용할 수 있는 범위라면 어디서나 접근 가능한 공개된 멤버로, 주로 private 멤버를 해당 클래스 외부에서 사용하게 하기 위한 멤버함수를 정의할 때 사용한다.
- protected: 상속에 의한 사용 가능.



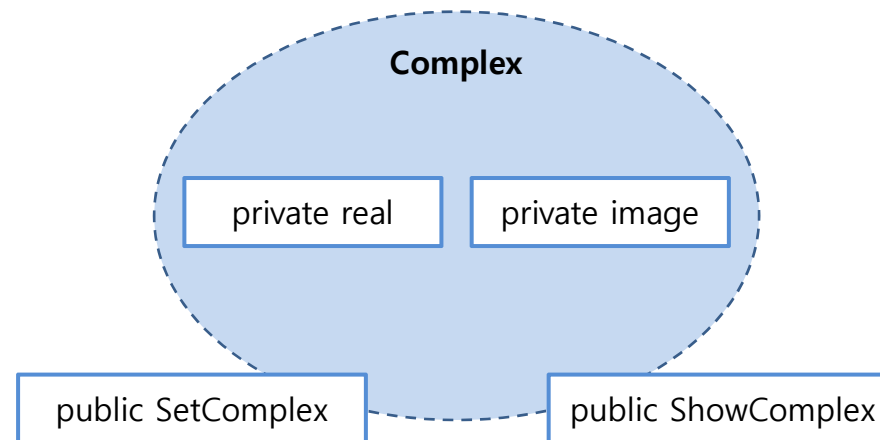
# 01 클래스의 이해

## 예제)

- 복소수를 Complex라는 이름의 클래스로 설계해 보자.
- 클래스는 새로운 자료형이 되고 데이터는 멤버변수가, 데이터를 처리할 함수는 멤버함수가 된다.

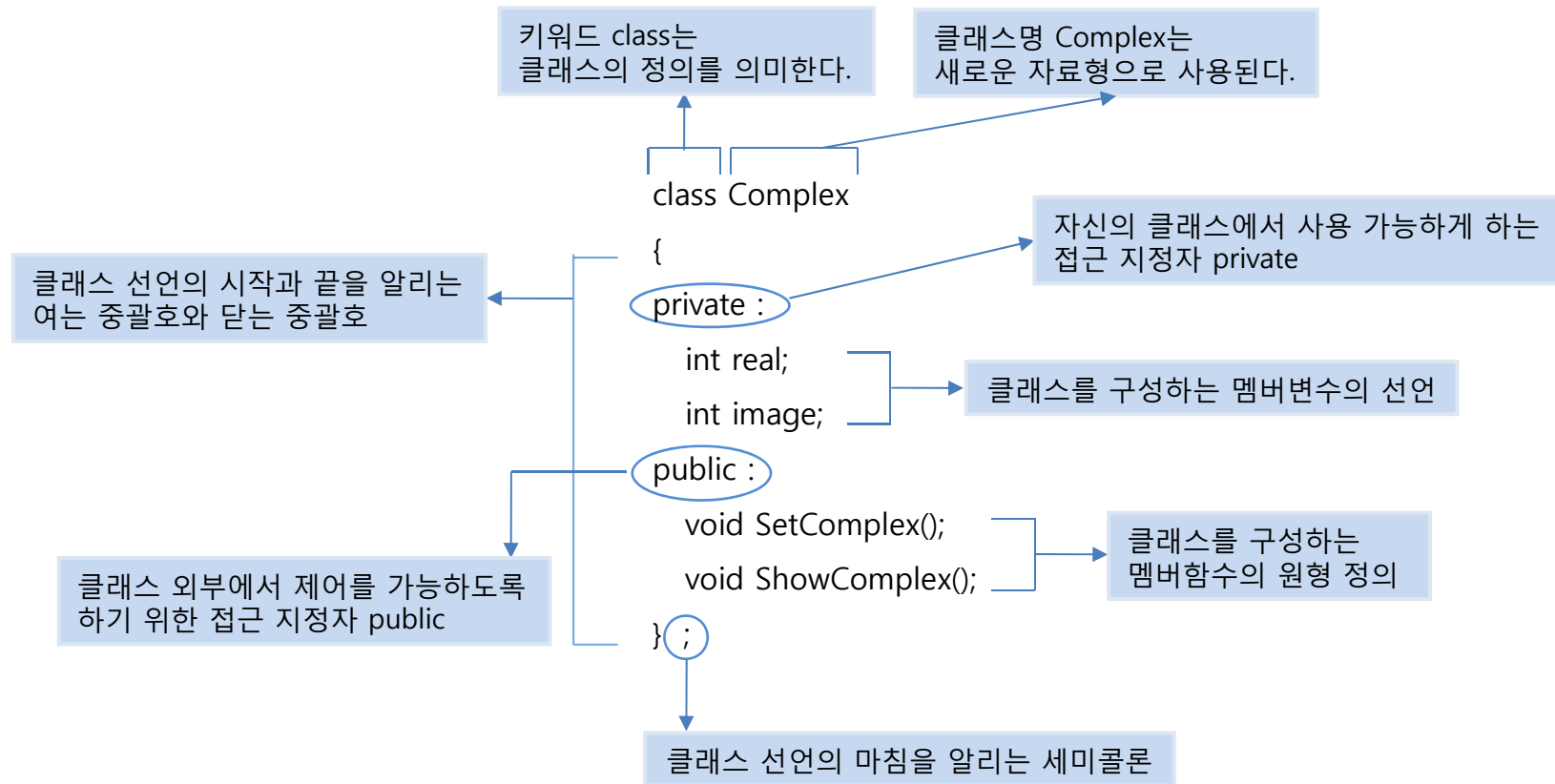
새로운 자료형(클래스) = 데이터의 저장(멤버변수) + 데이터를 처리할 함수(멤버함수)

- 실수를 저장하기 위한 멤버변수를 real, 허수를 저장하기 위한 멤버변수를 image라고 하자.  
데이터 은닉에 입각해서 멤버변수의 접근 지정자는 private로 선언하는 것이 일반적이다.
- 두 멤버변수에 값을 설정(SetComplex)하거나 보기(ShowComplex) 위한 멤버함수를 클래스 외부에서 접근할 수 있도록 public 접근 지정자를 이용해 공개한다.



[그림 10-1] Complex 클래스 설계

# 01 클래스의 이해



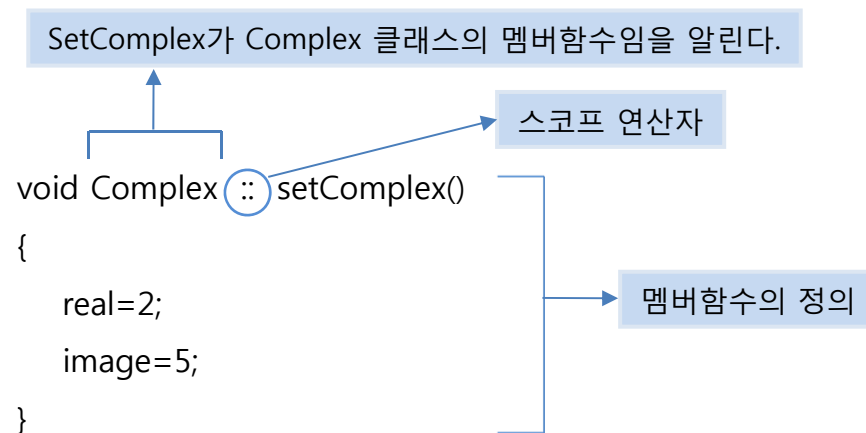
[그림 10-2] Complex 클래스 구현

# 01 클래스의 이해

## ■ 클래스의 멤버함수 정의(구현)

### ■ 멤버함수의 특징

- ① 멤버함수를 정의할 때에는 그 함수가 어느 클래스에 소속되는지 나타내기 위해 함수명 앞에 클래스명을 명시해야 한다. 이때 클래스명 다음에 **스코프 연산자(::)**를 사용한다.
- ② 클래스의 멤버함수를 정의하는 목적은 클래스 내에 정의된 private 멤버에 접근해서 이를 다루기 위해서다. 그래서 멤버함수를 '**메소드**'라고도 한다.



[그림 10-3] SetComplex 에 대한 정의

# 01 클래스의 이해

## ■ 객체 선언과 멤버 참조

- 객체는 클래스를 실체(instance)화한 것이다. 그래서 객체를 '인스턴스'라고도 한다.

[class] 클래스명 객체명1, 객체명2, ..., 객체명n;

객체 선언 기본 형식

- Complex 클래스를 이용해서 x와 y라는 객체 2개를 생성한 예.

Complex	x, y;
↓	↓
클래스명	객체(인스턴스)

# 01 클래스의 이해

## ■ 클래스 멤버의 접근 방법

- 멤버를 사용하려면 구조체처럼 . 과 -> 같은 멤버참조 연산자를 사용한다.

. 연산자를 이용한 클래스 멤버 접근 방법

```
객체명.멤버변수;  
객체명.멤버함수();
```

- 복소수를 구현한 Complex 클래스로 객체를 생성한 후 멤버함수를 호출하는 예를 보자.

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

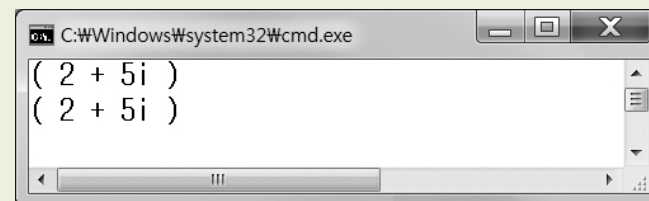
- SetComplex는 멤버함수이므로 함수명 앞에 객체명과 멤버참조 연산자를 함께 기술해야 한다.

```
SetComplex();    // 에러 ----- 잘못된 예(X)  
x.SetComplex();  // 객체명.멤버함수(); ----올바른 예(O)
```

## 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 }
```



# 01 클래스의 이해

---

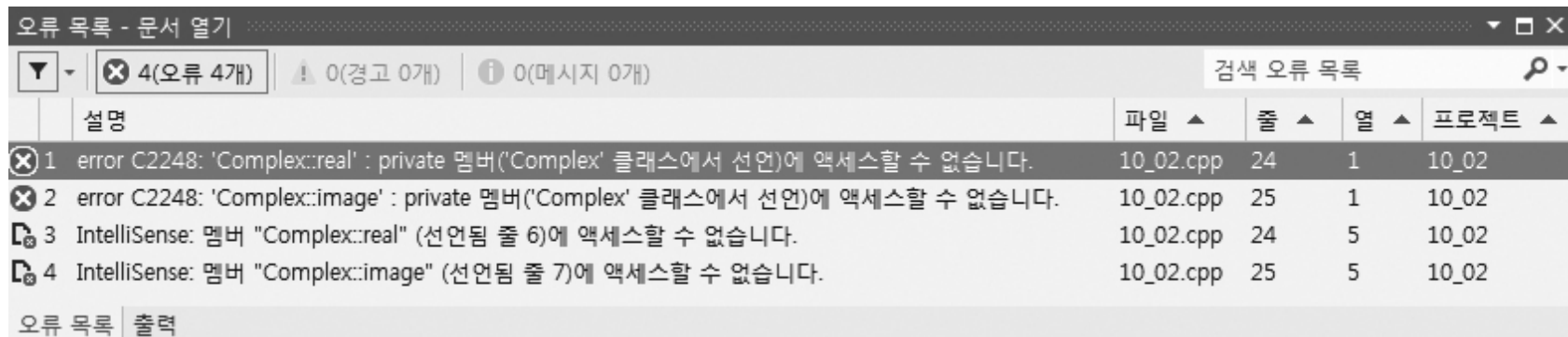
## ■ 클래스의 접근 지정자, private/public

- private 접근 지정자
  - ① 접근 지정자가 생략되면 기본(default)으로 private가 적용된다. private:를 명시적으로 기술할 수도 있다.
  - ② private 멤버의 사용 범위는 소속된 클래스 내의 멤버함수로 국한된다.
  - ③ 일반적으로 멤버변수를 private로 설정한다.
- public 접근 지정자
  - ① public 멤버로 지정하려면 public:을 명시적으로 기술해야 한다.
  - ② private 멤버변수를 처리하기 위한 목적으로 작성하는 멤버함수는 일반적으로 public 멤버로 설정한다.

## 예제 10-2. private 멤버 성격 파악하기(10\_02.cpp)

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real;
    int image;
public:
    void SetComplex();
    void ShowComplex();
};
void Complex::SetComplex() {
    real = 2;
    image = 5;
}
```

```
void Complex::ShowComplex()
{
    cout << "( " << real << " + " << image << "i )" << endl;
}
void main()
{
    Complex x;
    x.real = 5;    // private 멤버데이터에 접근 ==> 컴파일 에러
    x.image = 10; // private 멤버데이터에 접근 ==> 컴파일 에러
    x.ShowComplex();
}
```





# 01 클래스의 이해

---

- 예제[10-2]의 에러를 해결하는 방법은?
  - ① 누구나 real과 image에 접근하여 사용할 수 있도록 이 두 멤버변수의 접근 지정자를 public으로 변경한다.
  - ② private 멤버변수에 접근할 수 있는 public 멤버함수를 정의하여 그 멤버함수를 호출한다.

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

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

class Complex {
private:
    int real;
    int image;
public:
    void SetComplex();
    void ShowComplex();
    void SetReal(int r);
    void SetImage(int i);
};

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

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

void Complex::SetReal(int r) {
    real = r;
}

void Complex::SetImage(int i) {
    image = i;
}

void main() {
    Complex x;
    x.SetReal(5);
    x.SetImage(10);
    x.ShowComplex();
}
```

# 01 클래스의 이해

## ■ 클래스 내부에 멤버함수 정의하기

- 인라인 함수
  - 인라인 함수는 주로 함수의 정의가 짧을 때 사용하고, 인라인 함수를 정의할 때는 함수 선언 앞에 **inline**이라는 예약어를 써 준다 (함수 호출의 시간지연을 줄이기 위해서 함수가 실질적으로 호출되지 않고 프로그램 코드 사이에 컴파일된 함수 코드가 삽입됨).

### 인라인 함수 기본 형식

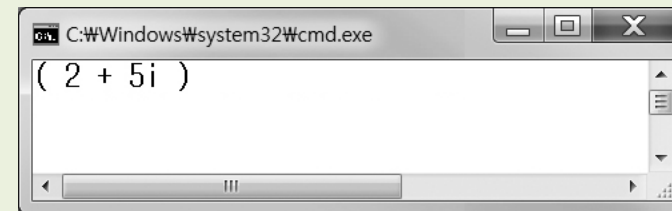
```
inline 자료형 함수명 (매개변수리스트)
{
    변수 선언;
    문장;
    return (결과값);
}
```

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

## 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()    // 자동 인라인 함수
10     {
11         real=2;
12         image=5;
13     }
14     void ShowComplex();
15 };
16
17
18 inline void Complex::ShowComplex() // 인라인 함수
19 {
20     cout<<"( " <<real <<" + " <<image <<"i )" <<endl ;
21 }
```

```
22 void main()
23 {
24     Complex x;
25
26     x.SetComplex();
27     x.ShowComplex();
28 }
```



# 01 클래스의 이해

## ■ const 상수와 const 멤버함수

- 상수를 정의하는 방법으로 예약어 const를 사용한다 (const 상수).  
일반 변수 선언과 유사하지만 반드시 초기값을 주어야 한다.

```
const 자료형 변수명 = 초기값;
```

const 상수 기본 형식

- const 상수는 매크로 상수(#define 문으로 정의하는 상수)와 동일한 목적으로 사용된다.  
하지만 매크로 상수는 자료형을 명시하지 않지만 const 상수는 자료형을 명시한다는 점이 다르다.
- 함수 내부에서 멤버변수 값을 변경하지 말아야 할 때 const 예약어를 사용한다 (const 멤버함수).

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

## 예제 10-5. const 멤버함수 사용하기(10\_05.cpp)

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

class Complex
{
private:
    int real;
    int image;
public:
    void SetComplex();
    void ShowComplex() const; // const 멤버함수 원형
};

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

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

void main()
{
    Complex x;
    x.SetComplex();
    x.ShowComplex();
}
```

# 01 클래스의 이해

## ■ 함수의 오버로딩

- 함수의 시그니처
  - 함수를 구분하기 위한 요소를 시그니처(signature)라 한다.
    - ❶ 함수명
    - ❷ 매개변수의 개수
    - ❸ 매개변수의 자료형
- 함수의 오버로딩
  - 함수명은 동일하게 사용되 함수를 호출할 때 모호하지 않도록 매개변수의 개수나 매개변수의 자료형을 다르게 하여 함수를 정의하는 것 → 함수의 다형성(polymorphism)
- 함수의 오버로딩이 필요한 이유
  - 같은 의미로 사용하는 함수를 모두 다른 이름으로 정의한다면 프로그램을 작성할 때마다 함수명을 개별적으로 지어주고 외워야 할 것이다. 예를 들어 절댓값을 구하는 함수를 다음과 같이 정의한다면?

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

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

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

```
int myabs(int num) {
    if (num<0)
        num = -num;
    return num;
}
```

```
double myabs(double num) {
    if (num<0)
        num = -num;
    return num;
}
```

```
long myabs(long num) {
    if (num<0)
        num = -num;
    return num;
}
```

```
void main()
{
    int a = -10;
    cout << a << "의 절댓값은-> " << myabs(a) << endl;

    double b = -3.4;
    cout << b << "의 절댓값은-> " << myabs(b) << endl;

    long c = -20L;
    cout << c << "의 절댓값은-> " << myabs(c) << endl;
}
```



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

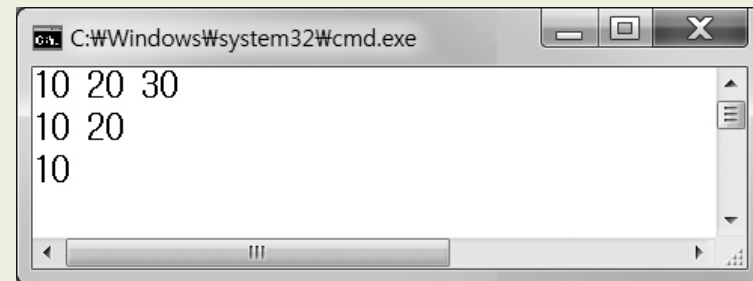
```
#include <iostream>
using namespace std;

void print(int x, int y, int z) {
    cout<<x<<" "<<y<<" "<<z<<endl;
}

void print(int x, int y) {
    cout<<x<<" "<<y<<endl;
}

void print(int x) {
    cout<<x<<endl;
}

void main() {
    print(10, 20, 30);
    print(10, 20);
    print(10);
}
```



# 01 클래스의 이해

---

## ■ 함수의 기본 매개변수

- 함수의 형식 매개변수에 값을 미리 설정할 수 있는데, 이렇게 형식 매개변수에 값을 설정해 놓은 것을 기본 매개변수라 한다.

```
void print(int x=10, int y=20, int z=30);
```

- 기본 매개변수는 함수를 호출할 때 대응되는 실 매개변수를 모두 기술하지 않아도 자동으로 적용되어 사용할 수 있는 매개변수다. 기본 매개변수는 함수의 선언부(원형)에서 지정해준다.

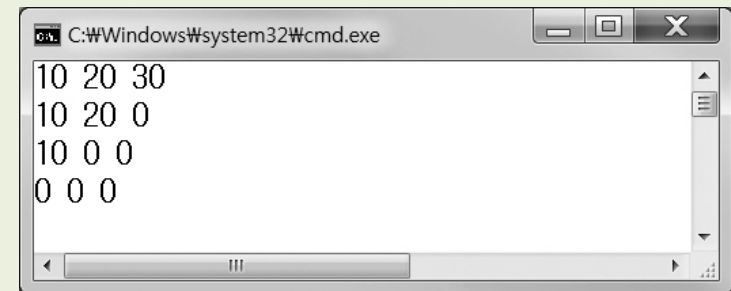
## 예제 10-10. 함수의 매개변수에 기본값 지정하기(10\_10.cpp)

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

void print(int x=0, int y=0, int z=0);

void main()
{
    print(10, 20, 30);
    print(10, 20);
    print(10);
    print();
}

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



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of the C++ program, which consists of four lines of space-separated integers: "10 20 30", "10 20 0", "10 0 0", and "0 0 0". The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

## 02 생성자와 소멸자

### ■ 생성자(constructor)의 의미와 특징

- 클래스도 객체를 생성할 때 기본 자료형처럼 초기값을 줄 수 있는데, 이를 가능하게 하는 것이 생성자다.
- 생성자의 특징
  - ① 생성자는 특별한 멤버함수다.
  - ② 생성자명은 클래스명과 동일하다.
  - ③ 생성자는 반환값의 자료형을 지정하지 않는다.
  - ④ 생성자의 호출은 명시적이지 않다.
  - ⑤ 생성자는 객체를 선언(생성)할 때 컴파일러에 의해 자동으로 호출된다.
  - ⑥ 객체의 초기화란 멤버변수의 초기화를 의미한다.
- 프로그래머가 생성자를 명시적으로 만들지 않으면 C++ 컴파일러는 매개변수가 없는 생성자를 자동으로 만든다. 이러한 생성자를 '기본 생성자(default constructor)'라고 한다.

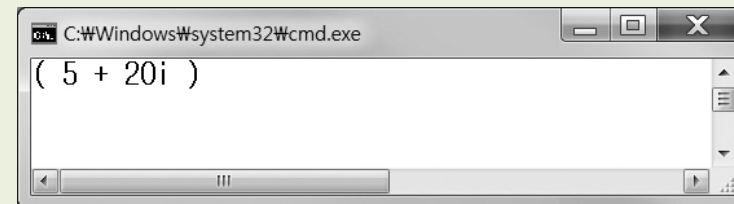
`Complex x; // 기본 생성자 호출`

- 객체를 생성할 때 생성자 재정의의 통해 멤버변수를 특정한 값으로 초기화 해줄 수 있다.

## 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 }
```



## 예제 10-12. 다양한 초깃값의 매개변수를 이용하는 생성자 작성하기(10\_12.cpp)

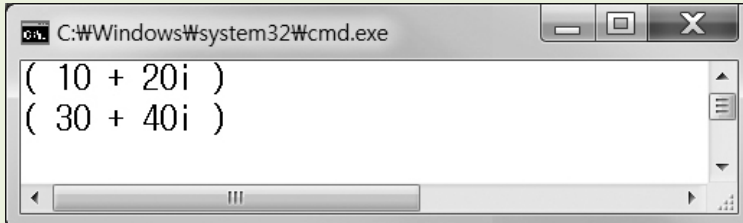
```
#include <iostream>
using namespace std;

class Complex
{
private :
    int real;
    int image;
public :
    Complex(int r, int i);      // 매개변수를 가지는 생성자
    void ShowComplex() const;
};

Complex::Complex(int r, int i) // 매개변수를 가지는 생성자
{
    real=r;
    image=i;
}
```

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

void main()
{
    Complex x(10, 20);
    Complex y(30, 40);
    // Complex z;
    x.ShowComplex();
    y.ShowComplex();
}
```



```
C:\Windows\system32\cmd.exe
( 10 + 20i )
( 30 + 40i )
```

## 02 생성자와 소멸자

### ■ 생성자 오버로딩

- 매개변수가 없는 기본 생성자는 C++ 컴파일러가 제공해 준다.

그런데, 프로그래머가 매개변수를 갖는 기본 생성자를 만들어주면 컴파일러는 더 이상 기본 생성자를 제공하지 않는다. 예를 들어 [예제10-12]에서 28행의 주석문을 제거하면 다음과 같은 에러가 발생한다.

오류 목록

오류 2개

경고 0개

메시지 0개

검색 오류 목록

설명	파일	줄	열	프로젝트
1 error C2512: 'Complex': 사용할 수 있는 적절한 기본 생성자가 없습니다.	10_12.cpp	28	1	10_12
2 IntelliSense: "Complex" 클래스의 기본 생성자가 없습니다.	10_12.cpp	28	10	10_12

- 컴파일 에러를 없애려면 프로그래머가 기본 생성자를 하나를 더 추가해서 정의해야 한다.
- 동일한 이름의 함수를 여러 번 정의할 수 있는 함수의 오버로딩은 생성자에도 적용된다.  
매개변수의 개수나 자료형을 달리해서 여러 번 정의할 수 있는데, 이를 생성자 오버로딩이라고 한다.

## 예제 10-13. 생성자 오버로딩하기(10\_13.cpp)

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

class Complex
{
private:
    int real;
    int image;
public:
    Complex();           // 사용자 정의 기본 생성자 원형
    Complex(int r, int i); // 생성자 오버로딩
    void ShowComplex() const;
};

Complex::Complex()      // 사용자 정의 기본 생성자
{
    real = 0;
    image = 0;
}
```

```
Complex::Complex(int r, int i) // 매개변수를 가지는 생성자 정의
{
    real = r;
    image = i;
}

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

void main()
{
    Complex x(10, 20);
    Complex y(30, 40);
    Complex z;           // 사용자 정의 기본 생성자 호출

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



## 02 생성자와 소멸자

### ■ 생성자의 기본 매개변수 값 지정하기

- 생성자에 기본 매개변수 값을 생성자 선언문(원형)에 다음과 같이 설정할 수 있다.

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

- 기본 매개변수 값을 기술하면 생성자를 한 번만 정의해서 매개변수 값을 다양한 형태로 호출 할 수 있다.

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

### ■ 생성자의 콜론 초기화

- 생성자의 헤더 부분에서 멤버변수의 초기값을 설정할 수 있는데, 이때 콜론 초기화가 사용된다.

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

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

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

class Complex
{
private:
    int real;
    int image;
public:
    Complex(int r = 0, int i = 0);    // 기본 매개변수
    void ShowComplex() const;
};

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

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

void main()
{
    // 기본 매개변수를 이용해 다양한 형태로 호출
    Complex x(10, 20);
    Complex y(30);
    Complex z;

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

## 예제 10-15. 생성자 콜론 초기화(10\_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;
};

// 콜론을 이용한 멤버변수 초기화
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(30);
    Complex z;

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

## 02 생성자와 소멸자

### ■ 소멸자의 의미와 특징

- 소멸자는 생성자의 반대되는 개념이다.
  - ① 생성자(constructor)는 객체가 생성될 때 자동 호출되고, 소멸자(destructor)는 객체가 소멸될 때 자동으로 호출된다.
  - ② 생성자가 객체를 초기화하기 위한 멤버함수라면, 소멸자는 객체를 정리해 주는(리소스를 해제하는 작업 등) 멤버함수다.
- 소멸자의 특징
  - ① 소멸자 함수는 멤버함수다.
  - ② 소멸자 함수명도 생성자처럼 클래스명을 사용한다.
  - ③ 소멸자 함수는 생성자 함수와 구분하려고 함수명 앞에 ~ 기호를 붙인다.
  - ④ 소멸자도 반환 자료형을 지정하지 않는다.
  - ⑤ 소멸자의 호출은 명시적이지 않다.
  - ⑥ 소멸자는 객체 소멸 시 자동 호출된다.
  - ⑦ 소멸자는 전달 매개변수를 지정할 수 없다.
  - ⑧ 소멸자는 전달 매개변수를 지정할 수 없으므로 오버로딩 할 수 없다.

## 예제 10-16. 소멸자 정의하기(10\_16.cpp)

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

class Complex
{
private:
    int real;
    int image;
public:
    Complex(int r = 0, int i = 0);
    ~Complex();           // 소멸자 원형
    void ShowComplex() const;
};

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

```
Complex::~~Complex()    // 소멸자 정의
{
    cout << "소멸자가 호출된다. \n";
}

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

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

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

# Homework

---

- Chapter 10 Exercise: 3, 4, 5, 6