■ 접근 제어 지시자

- 접근제어 지시자는 구조체가 클래스로 탈바꿈하도록 돕는 문법
- 제작자가 사용자가 이용할 수 있는 코드를 제한하는 용도로 활용
- 클래스의 특정 요소에 사용자가 접근할 수 없게 하기 위해 접근제어 지시자를 활용

접근제어 지시자	
✓ **public**	멤버에 관한 모든 외부 접근이 허용됩니다.
✓ **protected**	멤버에 관한 모든 외부 접근이 차단됩니다. 단, 상속 관계에 있는 파생클래스에서의 접근은 허용됩니다.
✓ **private**	외부 접근 뿐만 아니라 파생 클래스로부터의 접근까지 모두 차단됩니다. 클래스를 선언할 때 별도로 접근 제어 지시자를 기술하지 않으면 private 로 간주합니다.

- 객체 내부 멤버 변수의 임의 접근 차단

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

// 제작자 코드
class CMyData
{
    // 기본 접근 제어 지시자는 'private'이다
    int m_nData;

public:
    int GetData() { return m_nData;  }
    void SetData(int nParam) { m_nData = nParam;  }
};
```

```
// 사용자 코드 #1
int main()
{
    CMyData data;
    data.SetData(10);
    cout << data.GetData() << endl;
    return 0;
}
```

```
// 사용자 코드 #2
int main()
{
    CMyData data;
    data.m_nData = 10; // 허용되지 않음
    cout << data.GetData() << endl;
    return 0;
}
```

✓ CMyData 클래스의 m_nData에 대한 접근제어 지시자는 private
✓ 사용자 코드에서 멤버 접근 연산자를 이용해 멤버 변수에 임의 접근할 수 없습니다.
✓ 사용자의 임의 접근을 차단해서 제작자가 얻는 이득은?
  사용자가 임의로 멤버 변수의 값을 변화시키는 것을 통제할 수 있다.
✓ 사용자 코드1에서 GetData()/SetData() 메서드를 이용해서 m_nData의 값을 변경할 수 있도록 허용
✓ 두 함수는 사용자가 m_nData 의 값을 읽거나 쓸 수 있는 유일한 통로

● 객체 내부 멤버 변수의 임의 접근 차단

```
public:
    int GetData() { return m_nData;  }
    void SetData(int nParam) { m_nData = nParam;  }
```

✓ 멤버 함수의 경우에는 private 멤버에 접근하는 것이 자유롭다.
  → 접근제어 지시자의 영향을 받지 않는다.
✓ 생성자도 접근제어 지시자의 영향을 받는다.

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

// 제작자 코드
class CMyData
{
    // 기본 접근 제어 지시자는 'private'이다.
    CMyData() { }  // 생성자를 private로 선언
    int m_nData;

public:
    int GetData() { return m_nData;  }
    void SetData(int nParam) { m_nData = nParam;  }
};

// 사용자 코드
int main()
{
    CMyData data;
    /*
       생성자 함수를 private으로 선언한다면 사용자 코드에서 단지 객체를 선언하는 것으로도
       컴파일 오류가 발생
    */
    return 0;
}
```

15. 생성자와 소멸자

- 생성자(Constructor)와 소멸자(Destructor)는 클래스 객체가 생성 및 소멸될 때 '자동으로' 호출되는 함수
- 두 함수의 가장 큰 특징은 함수임에도 불구하고 '반환 형식이 없다'는 것과 함수 이름이 클래스 이름과 같다는 점
- 소멸자의 이름앞에는 ~를 붙인다.
- 생성자는 다중 정의할 수 있고, 소멸자는 하나만 유일.

```
클래스이름();  // 생성자
~클래스이름(); // 소멸자
```

- 디폴드 생성자 : 매개변수가 하나도 없는 생성자
- 클래스 제작자가 디폴드 생성자와 소멸자를 기술하지 않아도 컴파일러가 알아서 넣는다!
  → 생성자와 소멸자가 없는 클래스는 없다!

```
#include <iostream>
using namespace std;
class CTest
{
public:
    CTest()     // 생성자
    {
        cout << "CTest::CTest() " << endl; // 객체 생성시점 확인
    }
    ~CTest()    // 소멸자
    {
        cout << "CTest::~CTest() " << endl; // 객체 소멸시점 확인
    }
};
```

```
// #1
int main()
{
    cout << "Begin " << endl;
    CTest a;
    cout << "End " << endl;
    return 0;
}
```

```
// #2
CTest a;
int main()
{
    cout << "Begin " << endl;
    cout << "End " << endl;
    return 0;
}
```

✓ CTest 클래스의 인스턴스인 a를 선언 → 선언된 시점이 객체가 생성되는 시점, 생성자 호출
✓ #1에서 a가 main() 함수에 속한 지역변수
✓ 지역변수는 특성상 선언된 블록 범위가 끝나면 자동으로 소멸 (실행결과 확인!!!)
✓ #2에서 CTest 클래스의 인스턴스인 a를 전역변수로 선언 (실행결과 확인!!)
✓ 전역변수로 선언한 클래스의 생성자가 main() 함수보다 먼저 호출됨!

● 생성자 소멸자 호출순서 확인!

```
#include <iostream>
using namespace std;
class CTest
{
public:
    CTest()     // 생성자
    {
        cout << "CTest::CTest() " << endl; // 생성자 호출 확인
    }
    ~CTest()    // 소멸자
    {
        cout << "CTest::~CTest() " << endl; // 소멸자 호출 확인
    }
};
int main()
{
    cout << "Begin " << endl;
    CTest a;
    cout << "Before b" << endl;
    CTest b;
    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인! (생성자와 소멸자 호출시점)
✓ main() 함수의 마지막 구문이 실행된 후 내부에 선언된 지역변수들이 소멸됨.

● 정리해보면...

✓ main() 함수가 호출되기 전에 생성자가 호출될 수 있다.
✓ 생성자는 다중 정의할 수 있다.
✓ 소멸자는 다중 정의할 수 없다.
✓ main() 함수가 끝난 후에 소멸자가 호출될 수 있다.
✓ 생성자와 소멸자는 생략할 수 있으나 생략할 경우 컴파일러가 만들어 넣는다.

● 생성자 다중정의 했을 때 호출확인

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

class CTest
{
    int m_nData; // 멤버변수 private으로 선언
public:
    CTest() : m_nData(0)    // 디폴트 생성자(매개변수가 없음) m_nData(0) 초기화 목록 멤버변수 초기화
    {
        cout << "CTest::CTest() " << endl; // 생성자 호출 확인
    }
    CTest(int nParam):m_nData(nParam)       // 다중정의된 생성자
    {
        cout << "CTest::CTest(int nParam) " << endl; // 생성자 호출 확인
    }
    ~CTest()    // 소멸자
    {
        // 멤버변수의 값을 함께 출력해본다.
        cout << "CTest::~CTest() " << m_nData << endl; // 소멸자 호출 확인
    }
};

int main()
{
    cout << "Begin " << endl;
    CTest a;  // 객체 선언 > 인스턴스 생성 > 생성자 호출(디폴트 생성자)
    cout << "Before b" << endl;
    CTest b(10); // 객체 선언 > 인스턴스 생성 > 생성자 호출
    cout << "End " << endl;
    return 0;
}
```

✓ CTest a; 객체 선언은 매개변수가 없으므로 디폴트 생성자 호출
✓ CTest b(10); 객체 선언은 매개변수가 10 이 있으므로 다중정의된 생성자 호출

● 디폴트 생성자의 생략

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

class CTest
{
    int m_nData; // 멤버변수 private으로 선언
public:
    CTest(int nParam):m_nData(nParam)       // 다중정의된 생성자
    {
        cout << "CTest::CTest(int nParam) " << endl; // 생성자 호출 확인
    }
    ~CTest()    // 소멸자
    {
        // 멤버변수의 값을 함께 출력해본다.
        cout << "CTest::~CTest() " << m_nData << endl; // 소멸자 호출 확인
    }
};

int main()
{
    cout << "Begin " << endl;
    CTest a(20);  // 객체 선언 > 인스턴스 생성 > 생성자 호출
    cout << "Before b" << endl;
    CTest b(10); // 객체 선언 > 인스턴스 생성 > 생성자 호출
    cout << "End " << endl;
    return 0;
}
```

✓ 새로운 생성자를 다중정의로 만들면서 디폴트 생성자를 생략할 수 있음
✓ CTest c; → 디폴트 생성자는 생략했으므로 디폴트 값 생성은 허용되지 않음으로 오류 발생!!

■ 동적 객체의 생성과 소멸

- 클래스의 인스턴스는 선언해서 생성 가능
- new 연산을 통해 동적으로 생성할 수 있음.
- 동적으로 생성된 객체는 delete 연산자로 삭제해야 함
- 동적 객체 생성 및 소멸을 시도할 경우 객체가 생성 및 소멸하는 시점을 코드에서 알 수 있음!
  → new 와 delete 연산자는 각각 생성자와 소멸자를 호출하기 때문
- new와 delete 연산자 사용

```
#include <iostream>
using namespace std;
class CTest
{
    int m_nData;
public:
    CTest() : m_nData(0)    // 디폴트 생성자
    {
        cout << "CTest::CTest() " << endl; // 생성자 호출 확인
    }
    CTest(int nParam):m_nData(nParam)       // 생성자
    {
        cout << "CTest::CTest(int nParam) " << endl; // 생성자 호출 확인
    }
    ~CTest()    // 소멸자
    {
        // 멤버변수의 값을 함께 출력해본다.
        cout << "CTest::~CTest() " << m_nData << endl; // 소멸자 호출 확인
    }
};
```

```
int main()
{
    cout << "Begin " << endl;
    CTest* pData = new CTest;   // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출(디폴트 생성자)
    CTest a;                    // 객체 선언 > 인스턴스 생성 > 생성자 호출(디폴트 생성자)
    cout << "Before b" << endl;
    // delete 연산자를 이용해 객체를 삭제
    delete pData;

    pData = new CTest(20);  // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출
    CTest b(10); // 객체 선언 > 인스턴스 생성 > 생성자 호출
    delete pData; // delete 연산자를 이용해 객체를 삭제
    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인! (생성자와 소멸자 호출시점)
✓ 객체를 지역변수로 선언했을 때, 동적 생성, delete 했을 때

● 배열로 생성한 객체 (class CTest 위의 소스 이용)

```
int main()
{
    cout << "Begin " << endl;
    // 배열로 new 연산을 수행할 수 있다.
    CTest* pData = new CTest[3]; // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출(디폴트 생성자)

    // 배열로 생성한 것은 반드시 배열로 삭제한다!
    delete[] pData;

    pData = new CTest(20);  // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출

    // delete 연산자를 이용해 객체를 삭제
    delete pData;
    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인! (생성자와 소멸자 호출시점)

```
int main()
{
    cout << "Begin " << endl;
    CTest* pData = new CTest[3]; // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출(디폴트 생성자)

    // delete 연산자를 이용해 객체를 삭제
    delete pData;

    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인! (생성자와 소멸자 호출시점)

✓ delete pData; 배열로 생성한 객체를 배열로 삭제하지 않는다면 첫 번째 요소 하나만 소멸하고 나머지는 그대로 메모리에 남아있음.

● 배열로 생성한 객체 + 멤버변수 초기화 (class CTest 위의 소스 이용)

```
int main()
{
    cout << "Begin " << endl;
    // 배열로 new 연산을 수행할 수 있다.
    CTest* pData = new CTest[3]{20,30,10}; // 객체 초기화
    // 객체 동적 선언 > 인스턴스 생성 > 생성자 호출 ( 매개변수 있으면 디폴트 생성자가 아님 )

    // 배열로 생성한 것은 반드시 배열로 삭제한다!
    delete[] pData;
    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인!

● 포인터 배열로 객체생성 및 초기화 (class CTest 위의 소스 이용)

```
int main()
{
    cout << "Begin " << endl;
    CTest* pData[10];   // 객체 포인터 배열 선언 ( 객체 생성아님!!)
    cout << "Before b" << endl;

    for (int i = 0; i < 10; i++)
        pData[i] = new CTest(3 * i); //  객체 동적 생성 및 초기화

    // delete 연산자를 이용해 객체를 삭제
    // delete [] pData; 이거 아님!!!
    for (int i = 0; i < 10; i++)
        delete pData[i]; //  배열에 동적으로 생성된 객체들을 삭제해야함!

    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인!

● 포인터 배열로 객체생성 및 초기화 + 범위기반 for문 이용 (class CTest 위의 소스 이용)

```
int main()
{
    cout << "Begin " << endl;
    CTest* pData[10];   // 객체 포인터 선언 ( 객체 생성아님!!)
    cout << "Before b" << endl;

    for (auto& n : pData)
        n = new CTest(10); // 여기서 동적으로 객체를 생성!!

    // delete 연산자를 이용해 객체를 삭제
    // delete [] pData; 이거 아님!!!
    for (auto& n : pData)
        delete n;       // 객체 생성한 만큼 객체를 삭제!!

    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인!

■ 참조 형식 멤버 초기화

- 클래스의 멤버 변수는 참조형식으로 선언할 수 있음.
- 참조자는 반드시 선언과 동시에 초기화 해야 함.(반드시 생성자 초기화 목록을 이용!)

```
#include <iostream>
using namespace std;
// 제작자 코드
class CRefTest
{
private:
    // 참조형 멤버는 객체가 생성될 때 반드시 초기화 해야 한다.
    int& m_nData;
public:
    // 참조형 멤버는 반드시 생성자 초기화 목록을 이용해 초기화한다.
    CRefTest(int &rParam) :m_nData(rParam)      // 생성자
    {
        cout << "CRefTest::CRefTest(int &rParam) " << m_nData << endl; // 생성자 호출 확인
    }
    ~CRefTest() // 소멸자
    {
        // 멤버변수의 값을 함께 출력해본다.
        cout << "CRefTest::~CRefTest() " << m_nData << endl; // 소멸자 호출 확인
    }
    int GetData() { return m_nData; }
};

// 사용자 코드
int main()
{
    cout << "Begin " << endl;

    int a(10);

    CRefTest t(a);

    cout << " GetData() : " << t.GetData() << endl;

    // 참조 원본인 a의 값을 수정
    a = 20;
    cout << " GetData() : " << t.GetData() << endl;

    cout << "End " << endl;
    return 0;
}
```

✓ 실행결과 반드시 확인! (생성자와 소멸자 호출시점)

✓ 생성자 목록이 아닌 CRefTest(int &rParam) { m_nData = rParam; } 초기화는 어떻게 될까?

● 참조 멤버가 있는 클래스 생성자 CRefTest() 의 비교

```
// #1
    CRefTest(int &rParam) :m_nData(rParam)      // 생성자
    {
        cout << "CRefTest::CRefTest(int &rParam) " << m_nData << endl; // 생성자 호출 확인
    }
// #2
    CRefTest(int rParam) :m_nData(rParam)       // 생성자
    {
        cout << "CRefTest::CRefTest(int &rParam) " << m_nData << endl; // 생성자 호출 확인
    }
```

✓ 생성자의 매개변수를 참조형식(int &rParam)이 아니라 int rParam 이라고 작성하면 함수의 매개변수는 함수 내부의 자동변수와 같으므로 함수가 반환될 때 매개변수는 소멸