Chapter 9

Index

클래스 템플릿 : 클래스를 찍어내는 모양자, 인스턴스를 선언할 때 typename 필수 기술

템플릿 특수화 : 특별한 형식이 있을 경우에 나머지 다른 형식들과 다른 코드를 적용하는 방법론

스마트 포인터 : 동적으로 할당한 인스턴스를 '자동으로' 삭제해주는 포인터

클래스 템플릿

- 클래스를 찍어내는 모양자

```
templete<typename T>
class 클래스이름{
  ......
}
```

▼ 클래스 템플릿 코드

```
/* 클래스 템플릿*/
#include "pch.h"
#include <iostream>
using namespace std;

// 'T'는 자료형이 된다.
template<typename T>
class CMyData
{
public:
    CMyData(T param) : m_Data(param) { }
    T GetData() const { return m_Data; }

    // 형식에 대한 변환자 제공
    operator T() { return m_Data; }
    void SetData(T param) { m_Data = param; }

private:
    // T 형식의 멤버 변수 선언
    T m_Data;
};

int main()
{
    CMyData<int> a(5);
    cout << a << endl;
    CMyData<double> b(123.45);
    cout << b << endl;

    // 문자열을 저장하기 위해 메모리를 동적으로 할당하지는 않는다.
    CMyData<char*> c("Hello");
    cout << c << endl;

    return 0;
}
```

템플릿 클래스 : 찍어서 만들어진 클래스

▼ 클래스 템플릿을 통한 배열 관리 코드

```
/* 클래스 템플릿을 통한 배열 관리 */
#include "pch.h"
#include <iostream>
using namespace std;

template<typename T>
```

```
class CMyArray
{
public:
    explicit CMyArray(int nSize) : m_nSize(nSize)
    {
        m_pData = new T[nSize];
    }

    ~CMyArray() { delete[] m_pData; }

    // 복사 생성자
    CMyArray(const CMyArray& rhs)
    {
        m_pData = new T[rhs.m_nSize];
        memcpy(m_pData, rhs.m_pData, sizeof(T) * rhs.m_nSize);
        m_nSize = rhs.m_Size;
    }

    // 대입 연산자
    CMyArray& operator=(const CMyArray& rhs)
    {
        if (this == &rhs)
            return *this;

        delete m_pData;
        m_pData = new T[rhs.m_nSize];
        memcpy(m_pData, rhs.m_pData, sizeof(T) * rhs.m_nSize);
        m_nSize = rhs.m_nSize;

        return *this;
    }

    // 이동 생성자
    CMyArray(CMyArray&& rhs)
    {
        operator = (rhs);
    }

    // 이동 대입 연산자
    CMyArray& operator=(const CMyArray&& rhs)
    {
        m_pData = rhs.m_pData;
        m_nSize = rhs.m_nSize;
        rhs.m_pData = nullptr;
        rhs.m_nSize = 0;
    }

    // 배열 연산자
    T& operator[](int nIndex)
    {
        if (nIndex < 0 || nIndex >= m_nSize)
        {
```

```
            cout << "ERROR: 배열의 경계를 벗어난 접근입니다." << endl;
            exit(1);
        }

        return m_pData[nIndex];
    }

    // 상수화된 배열 연산자
    T& operator[](int nIndex) const
    {
        return operator[](nIndex);
    }

    // 배열 요소의 개수를 반환
    int GetSize() { return m_nSize; }

private:
    T* m_pData = nullptr;
    int m_nSize = 0;
};

int main()
{
    // int 자료형 배열
    CMyArray<int> arr(5);

    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;

    for (int i = 0; i < 5; ++i)
        cout << arr[i] << ' ';

    cout << endl;

    CMyArray<int> arr2(3);
    arr2 = arr;
    for (int i = 0; i < 5; ++i)
        cout << arr2[i] << ' ';

    cout << endl;

    return 0;
}
```

클래스 템플릿의 장점 : typename에 기본 형식 뿐만 아니라 클래스도 적용 가능

멤버 선언 및 정의

▼ 멤버 선언과 정의 분리 코드

```
/* 멤버 선언과 정의 분리 */
#include "pch.h"
#include <iostream>
using namespace std;

template<typename T>
class CTest
{
public:
    // 생성자 선언
    CTest();
    T TestFunc();

protected:
    // 정적 멤버 데이터 선언
    static T m_Data;
};

// 생성자 정의
template<typename T>
CTest<T>::CTest()
{
}

// 멤버 함수 정의
template<typename T>
T CTest<T>::TestFunc()
{
    return m_Data;
}

// 정적 멤버 변수 정의
template<typename T>
T CTest<T>::m_Data = 15;

int main()
{
    CTest<double> a;
    cout << a.TestFunc() << endl;

    return 0;
}
```

멤버의 정의를 클래스 선언 밖으로 빼내면 클래스 이름에 이어 <형식> 필수 기술

```
template<typename T>
클래스이름<T>::멤버함수이름()
{
}
```

템플릿 매개변수

▼ 함수처럼 선언해 사용하는 템플릿 매개변수 코드

```
/* 함수처럼 선언해 사용하는 템플릿 매개변수 */
#include "pch.h"
#include <iostream>
using namespace std;

// 템플릿 매개변수를 함수처럼 선언한다.
template<typename T,int nSize>
class CMyArray
{
public:
    // 이하 코드에 보이는 모든 nSize 변수는 템플릿 매개변수다.
    CMyArray() { m_pData = new T[nSize]; }
    ~CMyArray() { delete[] m_pData; }

    // 배열 연산자
    T& operator[](int nIndex)
    {
        if (nIndex < 0 || nIndex >= nSize)
        {
            cout << "ERROR: 배열의 경계를 벗어난 접근입니다." << endl;
            exit(1);
        }

        return m_pData[nIndex];
    }

    // 상수화된 배열 연산자
    T& operator[](int nIndex) const { return operator[](nIndex); }

    // 배열 요소의 개수를 반환
    int GetSize() { return nSize; }

private:
    T* m_pData = nullptr;
};

int main()
{
```

```
    CMyArray<int, 3>arr;
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << arr[i] << endl;

    return 0;
}
```

디폴트 값도 지정 가능

템플릿 특수화

- 특별한 형식이 있을 경우 나머지 다른 형식들과 전혀 다른 코드를 적용해야 할 때 사용

함수 템플릿 특수화

▼ 문자열과 다른 자료형을 나눠서 정의한 함수 템플릿 코드

```
/* 문자열과 다른 자료형을 나눠서 정의한 함수 템플릿 */
#include "pch.h"
#include <memory>
#include <iostream>
using namespace std;

template<typename T>
T Add(T a, T b) { return a + b; }

// 두 개의 변수가 모두 char* 형식이면 이 함수로 대체된다.
template< >
char* Add(char *pszLeft, char *pszRight)
{
    int nLenLeft = strlen(pszLeft);
    int nLenRight = strlen(pszRight);
    char* pszResult = new char[nLenLeft + nLenRight + 1];

    // 새로 할당한 메모리에 문자열을 복사한다.
    strcpy_s(pszResult, nLenLeft + 1, pszLeft);
    strcpy_s(pszResult + nLenLeft, nLenRight + 1, pszRight);

    return pszResult;
}

int main()
{
```

```
    int nResult = Add<int>(3, 4);
    cout << nResult << endl;

    char *pszResult = Add<char*>("Hello", "World");
    cout << pszResult << endl;
    delete[] pszResult;

    return 0;
}
```

두 매개변수와 반환 형식이 모두 같은 형식이어야 함

클래스 템플릿 특수화

▼ 클래스 템플릿 특수화 코드

```
/* 클래스 템플릿 특수화 */
#include "pch.h"
#include <iostream>
using namespace std;

template<typename T>
class CMyData
{
    CMyData(T param) : m_Data(param) { }

    T GetData() const { return m_Data; }
    void SetData(T param) { m_Data = param; }

private:
    T m_Data;
};

template< >
class CMyData<char*>
{
public:
    CMyData(char* pszParam)
    {
        int nLen = strlen(pszParam);
        m_Data = new char[nLen + 1];
    }

    ~CMyData() { delete[] m_Data; }
    char* GetData() const { return m_Data; }

private:
    char* m_Data;
};
```

```
int main()
{
    CMyData<char*> a("Hello");
    cout << a.GetData() << endl;

    return 0;
}
```

클래스 템플릿과 상속

```
template<typename T>
class CMyDataEx : public CMyData<T>
```

▼ 클래스 템플릿의 상속 코드

```
/* 클래스 템플릿의 상속 */
#include "pch.h"
#include <iostream>
using namespace std;

template<typename T>
class CMyData
{
public:

protected:
    T m_Data;
};

template<typename T>
class CMyDataEx : public CMyData<T>
{
public:
    T GetData() const { return m_Data; }
    void SetData(T param) { m_Data = param; }
};

int main()
{
    CMyDataEx<int> a;
    a.SetData(5);
    cout << a.GetData() << endl;
```

```
    return 0;
}
```

스마트 포인터

- 동적 할당한 인스턴스를 '자동'으로 삭제해주는 편리한 포인터

스마트 포인터	설명
auto_ptr	동적 할당한 인스턴스를 '자동'으로 삭제, 가장 오래 존재했던 스마트 포인터
shared_ptr	포인팅 횟수를 계수해서 0이 되면 대상을 삭제
unique_ptr	shared_ptr과 달리 한 대상을 오로지 한 포인터로만 포인팅, 하나의 소유자만 허용
weak_ptr	하나 이상의 shared_ptr 인스턴스가 소유하는 개체에 접근할 수 있게 하지만 참조 수로 계산하지 않는다. 특수한 경우에만 사용하고 거의 사용하지 않는다.

auto_ptr

- 쓰지 않는 것이 좋다.

▼ auto_ptr 스마트 포인터 사용 코드

```
/* auto_ptr 스마트 포인터 사용 */
#include "pch.h"
#include <memory>
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData() { cout << "CMyData()" << endl; }
    ~CMyData() { cout << "~CMyData()" << endl; }
};

int main()
{
    cout << "*****Begin*****" << endl;
    {
        // 속한 범위를 벗어나면 대상 객체는 자동으로 소멸한다.
        auto_ptr <CMyData> ptrTest(new CMyData);
    }

    cout << "******End******" << endl;
```

```
    return 0;
}
```

배열로 동적 할당시 문제 발생

▼ **얕은 복사를 실행한 auto_ptr 예제 코드**

```
/* 얕은 복사를 실행한 auto_ptr 예제 */
#include "pch.h"
#include <memory>
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData() { cout << "CMyData()" << endl; }
    ~CMyData() { cout << "~CMyData()" << endl; }
    void TestFunc() { cout << "CMyData::TestFunc()" << endl; }
};

int main()
{
    auto_ptr<CMyData> ptrTest(new CMyData);
    auto_ptr<CMyData> ptrNew;

    cout << "0x" << ptrTest.get() << endl;

    // 포인터를 대입하면 원본 포인터는 NULL이 된다.
    ptrNew = ptrTest;
    cout << "0x" << ptrTest.get() << endl;

    // 따라서 이 코드를 실행할 수 없다.
    ptrTest->TestFunc();

    return 0;
}
```

단순 대입 연산이 일반적인 경우처럼 ‘복사’가 아니라 ‘이동’이 된다.

shared_ptr

- 포인팅 횟수를 계산해서 0이 되면 대상을 삭제

▼ **포인팅 횟수를 계산하는 Shared_ptr 코드**

```
/* 포인팅 횟수를 계산하는 Shared_ptr */
#include "pch.h"

// shared_ptr 클래스를 사용하기 위함
#include <memory>

#include <iostream>
using namespace std;

class CTest
{
public:
    CTest() { cout << "CTest()" << endl; }
    ~CTest() { cout << "~CTest()" << endl; }
    void TestFunc() { cout << "TestFunc()" << endl; }
};

int main()
{
    cout << "*****Begin*****" << endl;
    shared_ptr<CTest> ptr1(new CTest);

    cout << "Counter: " << ptr1.use_count() << endl;
    {
        shared_ptr<CTest> ptr2(ptr1);

        // 한 대상을 한 포인터로 포인팅한다.
        cout << "Counter: " << ptr1.use_count() << endl;
        ptr2->TestFunc();
    }

    // 한 포인터가 소멸했으므로 포인팅 개수가 1 감소한다.
    cout << "Counter: " << ptr1.use_count() << endl;
    ptr1->TestFunc();
    cout << "******End******" << endl;

    // 결국 카운터가 0이 되면 대상 객체를 소멸시킨다.
    return 0;
}
```

▼ 배열로 대상을 삭제하는 RemoveTest( ) 함수 코드

```
/* 배열로 대상을 삭제하는 RemoveTest() 함수 */
#include "pch.h"
#include <memory>
#include <iostream>
using namespace std;
```

```
class CTest
{
public:
    CTest() { cout << "CTest()" << endl; }
    ~CTest() { cout << "~CTest()" << endl; }
    void TestFunc() { cout << "TestFunc()" << endl; }
};

void RemoveTest(CTest* pTest)
{
    cout << "RemoveTest()" << endl;

    // 대상을 배열로 삭제한다.
    delete[] pTest;
}

int main()
{
    cout << "*****Begin*****" << endl;

    // 대상 객체를 소멸할 함수를 별도로 등록한다.
    shared_ptr<CTest> ptr(new CTest[3], RemoveTest);
    cout << "******End******" << endl;

    return 0;
}
```

reset( ) 메서드를 호출해 즉시 삭제 가능

auto_ptr보다 shared_ptr가 좋다.

unique_ptr

▼ 한 포인터로만 포인팅하는 unique_ptr 코드

```
/* 한 포인터로만 포인팅하는 unique_ptr */
#include "pch.h"
#include <memory>
#include <iostream>
using namespace std;

class CTest
{
public:
    CTest() { cout << "CTest()" << endl; }
    ~CTest() { cout << "~CTest()" << endl; }
    void TestFunc() { cout << "TestFunc()" << endl; }
};
```

```
int main()
{
    unique_ptr<CTest> ptr1(new CTest);

    // 아래 코드들은 실행하면 모두 컴파일 오류가 발생한다.
    // unique_ptr<CTest> ptr2(ptr1);
    // ptr2 = ptr1;

    return 0;
}
```

하나의 대상에 오직 하나의 포인터만 존재할 수 있다.

weak_ptr

- shared_ptr이 가리키는 대상에 참조 형식으로 포인팅할 수 있다.
- 포인팅만 할 수 있어 거의 사용하지 않는다.

문제

1. **템플릿에서 다른 규칙의 코드를 적용하는 코드를 이용하는 경우 사용하는 문법**

   템플릿 특수화

2. **스마트 포인터 중 auto_ptr의 문제점**

   얕은 복사시 ‘이동’이됨

   배열에 사용 시 문제 발생