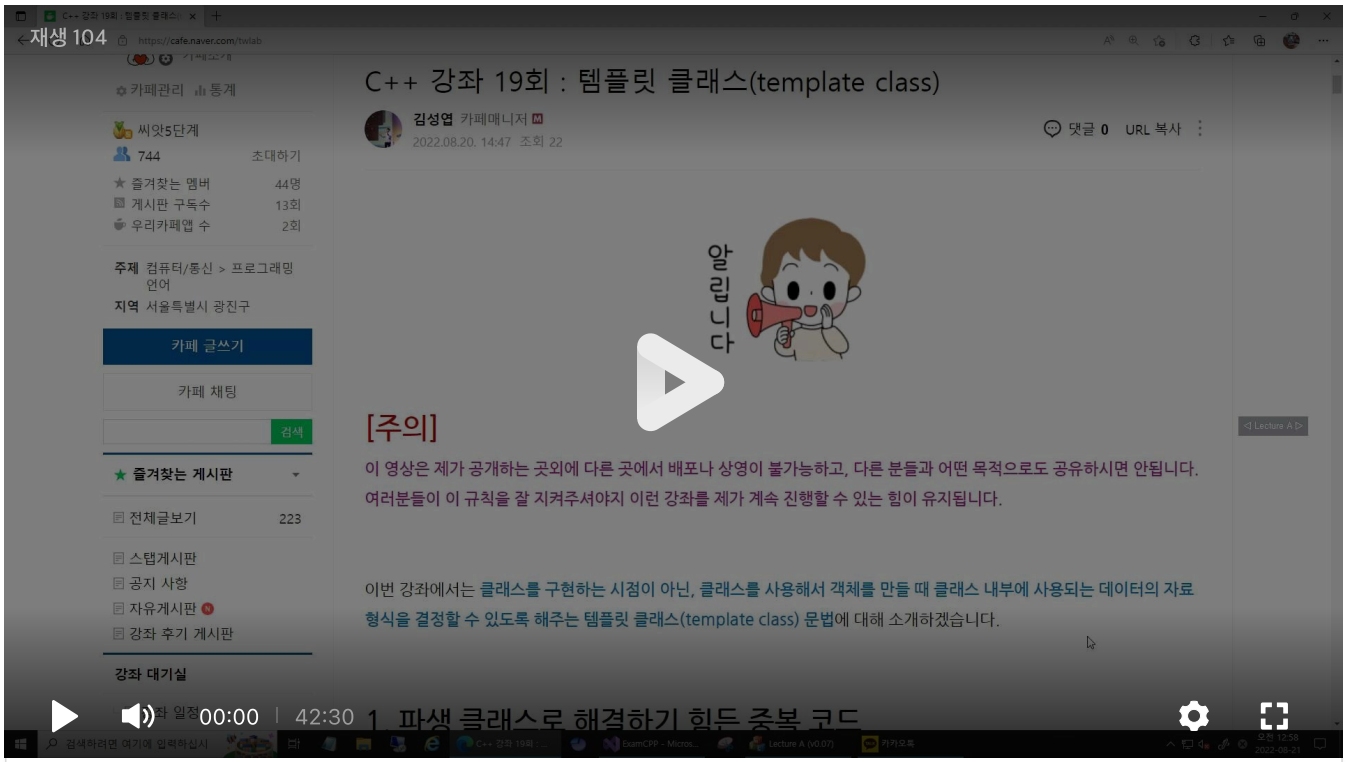
C++ 강좌 19회 : 템플릿 클래스(template class)

김성엽 카페매니저 + 구독 1:1 채팅
2022.08.20. 14:47 조회 348

[주의]

이 영상은 제가 공개하는 곳외에 다른 곳에서 배포나 상영이 불가능하고, 다른 분들과 어떤 목적으로도 공유하시면 안됩니다. 여러분들이 이 규칙을 잘 지켜주셔야지 이런 강좌를 제가 계속 진행할 수 있는 힘이 유지됩니다.

이번 강좌에서는 클래스를 구현하는 시점이 아닌, 클래스를 사용해서 객체를 만든 때 클래스 내부에 사용되는 데이터의 자료형식을 결정할 수 있도록 해주는 템플릿 클래스(template class) 문법에 대해 소개하겠습니다.



19회차. 템플릿 클래스

1. 파생 클래스로 해결하기 힘든 중복 코드

아래와 같이 **기능은 유사하지만 해당 클래스가 사용하는 데이터의 자료형이 다른 두 클래스**가 있다고 가정하겠습니다. 두 클래스 모두 값을 저장했다가 알려주는 기능을 가지고 있지만 IntData 클래스는 정숫값을 저장하거나 저장된 값을 알려주고 DoubleData 클래스는 실숫값을 저장하거나 저장된 값을 알려주는 클래스입니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

// 정숫값을 저장하거나 읽는 클래스
class IntData
{
private:
    int m_data;  // 값을 저장할 정수 변수

public:
    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(int a_data)
    {
        m_data = a_data;
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    int GetData()
    {
        return m_data;
    }
};

// 실숫값을 저장하거나 읽는 클래스
class DoubleData
{
private:
    double m_data;  // 값을 저장할 실수 변수

public:
    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(double a_data)
    {
        m_data = a_data;
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    double GetData()
    {
        return m_data;
    }
};

int main()
{
    // 정숫값과 실숫값을 사용할 객체를 선언한다.
    IntData i_temp;
    DoubleData d_temp;

    // i_temp 객체에 5를 저장한다. (i_temp::m_data 에 5가 저장된다.)
    i_temp.SetData(5);
    // d_temp 객체에 6.2를 저장한다. (d_temp::m_data 에 6.2가 저장된다.)
```

```
    d_temp.SetData(6.2);

    // i_temp 객체와 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "i_temp: " << i_temp.GetData() << endl;
    cout << "d_temp: " << d_temp.GetData() << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
i_temp: 5
d_temp: 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

위 예제에서 사용한 IntData 클래스와 DoubleData 클래스는 기능적으로 SetData, GetData 함수를 동일하게 가지고 있고 실제로 사용되는 코드도 유사하기 때문에 중복 코드로 보일 것입니다. 하지만 해당 함수가 서로 다른 자료형의 매개 변수를 사용하기 때문에 하나의 클래스로 합치면 아래와 같이 두 클래스의 내용이 그대로 합쳐져서 아무런 이득도 없고 사용하기만 더 불편하고 이 불편함이 오해가 되어 코드에 버그가 생길 확률만 높아질 것입니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

// 정숫값 또는 실숫값을 저장하거나 읽는 클래스
class UserData
{
private:
    int m_int_data;  // 값을 저장할 정수 변수
    double m_double_data;  // 값을 저장할 실수 변수

public:
    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetIntData(int a_data)
    {
        m_int_data = a_data;
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    int GetIntData()
    {
        return m_int_data;
    }

    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetDoubleData(double a_data)
    {
        m_double_data = a_data;
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    double GetDoubleData()
    {
        return m_double_data;
    }
};

int main()
{
    // 정숫값과 실숫값을 사용할 객체를 선언한다.
    UserData temp;

    // temp 객체에 5를 저장한다. (m_int_data 에 5가 저장된다.)
    temp.SetIntData(5);
    // temp 객체에 6.2을 저장한다. (m_double_data 에 6.2이 저장된다.)
    temp.SetDoubleData(6.2);

    // i_temp 객체와 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "temp(int): " << temp.GetIntData() << endl;
    cout << "temp(double): " << temp.GetDoubleData() << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
temp(int): 5
temp(double): 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

그리고 위 코드에 연산자 오버로딩 기술을 사용하면 좀더 사용하는 코드가 단순해 지겠지만, 그 효과는 위에 두 개로 유지하는 코드에도 동일하게 적용되기 때문에 **위 코드가 두 개의 클래스로 유지하는 코드보다 더 좋다고 이야기할 수는 없은 것 같습니다.** 그래서 포인터와 연산자 오버로딩을 능숙하게 사용하는 개발자라면 아래와 같이 코드를 구성할 것입니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

class UserData
{
private:
    // 값을 저장하기 위해 사용할 포인터
    // void *를 사용하는 것이 정석이지만 메모리 할당과 해제 그리고
    // 주소 연산시 char *를 사용하는 거이 더 편해서 char *로 설정함
    char *mp_data;
    // 저장된 값의 크기 (이 코드에서는 별로 의미 없음)
    int m_data_size;
public:
    UserData()  // 생성자
    {
        mp_data = NULL;
        m_data_size = 0;
    }

    ~UserData()  // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(void *ap_data, int a_data_size)
    {
        // 기존에 사용하던 메모리를 해제한다.
        if (mp_data) delete[] mp_data;

        // 정상적인 정보인지 체크한다.
        if (ap_data && a_data_size > 0) {
            // 저장할 크기에 맞춰서 메모리를 할당한다.
            mp_data = new char[a_data_size];
            // 할당된 메모리에 전달된 값을 저장한다.
            memcpy(mp_data, ap_data, a_data_size);
            // 전달된 크기를 저장한다.
            m_data_size = a_data_size;
        } else {
            // 정상적인 값이 전달되지 않았다면 내부 값을 초기화한다.
            mp_data = NULL;
            m_data_size = 0;
        }
    }

    // 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
    void operator =(int a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }
```

```
    // 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
    void operator =(double a_data)
    {
        // 실숫값을 저장한다.
        SetData(&a_data, sizeof(double));
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    void *GetData()
    {
        return mp_data;
    }

    // 정숫값을 반환하는 경우에 사용하는 함수
    operator int()
    {
        return *(int *)mp_data;
    }

    // 실숫값을 반환하는 경우에 사용하는 함수
    operator double()
    {
        return *(double *)mp_data;
    }
};

int main()
{
    // 정숫값과 실숫값을 사용할 객체를 선언한다.
    UserData i_temp, d_temp;

    // i_temp 객체에 5를 저장한다.
    i_temp = 5;
    // d_temp 객체에 6.2를 저장한다.
    d_temp = 6.2;

    // i_temp 객체와 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "i_temp: " << (int)i_temp << endl;
    cout << "d_temp: " << (double)d_temp << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
i_temp: 5
d_temp: 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

만약, 위 코드에서 메모리 재 할당률을 줄이고 싶다면 아래와 같이 코드를 구성해도 됩니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

// UserData 클래스가 사용할 데이터의 최대 크기
#define MAX_USER_DATA_SIZE  8

class UserData
{
private:
    // 값을 저장하기 위해 사용할 포인터
    // void *를 사용하는 것이 정석이지만 메모리 할당과 해제 그리고
    // 주소 연산시 char *를 사용하는 거이 더 편해서 char *로 설정함
    char *mp_data;
    // 저장된 값의 크기와 실제 할당된 크기
    int m_data_size, m_alloc_size;

public:
    UserData(int a_alloc_size = MAX_USER_DATA_SIZE)  // 생성자
    {
        mp_data = new char[a_alloc_size];
        m_data_size = 0;
        m_alloc_size = a_alloc_size;
    }

    ~UserData()  // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(void *ap_data, int a_data_size)
    {
        // 정상적인 정보인지 체크한다.
        if (a_data_size <= m_alloc_size) {
            // 할당된 메모리에 전달된 값을 저장한다.
            memcpy(mp_data, ap_data, a_data_size);
            // 전달된 크기를 저장한다.
            m_data_size = a_data_size;
        } else {
            // 정상적인 값이 전달되지 않았다면 복사된 크기 값을 초기화한다.
            m_data_size = 0;
        }
    }

    // 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
    void operator =(int a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }
```

```
    // 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
    void operator =(double a_data)
    {
        // 실숫값을 저장한다.
        SetData(&a_data, sizeof(double));
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    void *GetData()
    {
        return mp_data;
    }

    // 내부적으로 가지고 있는 데이터의 크기를 얻을 때 사용하는 함수
    int GetDataSize()
    {
        return m_data_size;
    }

    // 정숫값을 반환하는 경우에 사용하는 함수
    operator int()
    {
        return *(int *)mp_data;
    }

    // 실숫값을 반환하는 경우에 사용하는 함수
    operator double()
    {
        return *(double *)mp_data;
    }
};

int main()
{
    // 정숫값과 실숫값을 사용할 객체를 선언한다.
    UserData i_temp, d_temp;

    // i_temp 객체에 5를 저장한다.
    i_temp = 5;
    // d_temp 객체에 6.2를 저장한다.
    d_temp = 6.2;

    // i_temp 객체와 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "i_temp: " << (int)i_temp << endl;
    cout << "d_temp: " << (double)d_temp << endl;

    return 0;
}
```

사실 저는 이번 강좌에서 소개할 템플릿 클래스 문법보다는 위 코드를 더 선호하는 편입니다. 클래스를 구성하는 코드가 더 복잡하기는 하지만 결국 클래스를 사용하는 사람에게 분편함이 별로 없기 때문입니다.

하지만 이 클래스에 사용되는 자료형이 계속 늘어나게 되면 이 코드의 유지보수는 불편해 집니다. 예를 들어, 위 코드에 char 자료형을 추가해야 한다면 코드는 아래와 같이 변경될 것입니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

// UserData 클래스가 사용할 데이터의 최대 크기
#define MAX_USER_DATA_SIZE  8

class UserData
{
private:
    // 값을 저장하기 위해 사용할 포인터
    // void *를 사용하는 것이 정석이지만 메모리 할당과 해제 그리고
    // 주소 연산시 char *를 사용하는 거이 더 편해서 char *로 설정함
    char *mp_data;
    // 저장된 값의 크기와 실제 할당된 크기
    int m_data_size, m_alloc_size;

public:
    UserData(int a_alloc_size = MAX_USER_DATA_SIZE)  // 생성자
    {
        mp_data = new char[a_alloc_size];
        m_data_size = 0;
        m_alloc_size = a_alloc_size;
    }

    ~UserData()  // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(void *ap_data, int a_data_size)
    {
        // 정상적인 정보인지 체크한다.
        if (a_data_size <= m_alloc_size) {
            // 할당된 메모리에 전달된 값을 저장한다.
            memcpy(mp_data, ap_data, a_data_size);
            // 전달된 크기를 저장한다.
            m_data_size = a_data_size;
        } else {
            // 정상적인 값이 전달되지 않았다면 복사된 크기 값을 초기화한다.
            m_data_size = 0;
        }
    }

    // 대입 연산자를 사용해서 문자값이 전달된 경우에 호출되는 함수
    void operator =(char a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }
```

```
    // 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
    void operator =(int a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }

    // 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
    void operator =(double a_data)
    {
        // 실숫값을 저장한다.
        SetData(&a_data, sizeof(double));
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    void *GetData()
    {
        return mp_data;
    }

    // 내부적으로 가지고 있는 데이터의 크기를 얻을 때 사용하는 함수
    int GetDataSize()
    {
        return m_data_size;
    }

    // 문자값을 반환하는 경우에 사용하는 함수
    operator char()
    {
        return *(char *)mp_data;
    }

    // 정숫값을 반환하는 경우에 사용하는 함수
    operator int()
    {
        return *(int *)mp_data;
    }

    // 실숫값을 반환하는 경우에 사용하는 함수
    operator double()
    {
        return *(double *)mp_data;
    }
};

int main()
{
    // 문자값, 정숫값 그리고 실숫값을 사용할 객체를 선언한다.
    UserData c_temp, i_temp, d_temp;

    // c_temp 객체에 'A' 문자를 저장한다.
    c_temp = 'A';
    // i_temp 객체에 5를 저장한다.
```

```
    // i_temp 객체에 5를 저장한다.
    i_temp = 5;
    // d_temp 객체에 6.2를 저장한다.
    d_temp = 6.2;

    // c_temp 객체, i_temp 객체 그리고 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "c_temp: " << (char)c_temp << endl;
    cout << "i_temp: " << (int)i_temp << endl;
    cout << "d_temp: " << (double)d_temp << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
c_temp: A
i_temp: 5
d_temp: 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

물론 **연산자 오버로딩으로 함수 두 개만 추가**하면 되는 간단한 작업이라 사실 불편하다고 말하기 민망하긴 하지만 그래도 **클래스를 만든 사람이 변경해야 한다고 하면 클래스를 만든 사람 입장에서는 귀찮은 작업**이 되는건 사실입니다. 그리고 이 클래스가 라이브러리에 포함되어 있다면 클래스를 만든 사람이 위 작업을 하고 라이브러리를 다시 빌드해서 배포해야 하기 때문에 더 귀찮은 작업이 될 것입니다.

만약, 위 클래스의 구현 코드가 포함된 라이브러리를 변경할 수 없는 상황이거나 변경하기 귀찮은 경우에는 **상속 문법을 사용해서 클래스를 만든 사람이 아닌 클래스를 사용하는 사람에게 작업을 미룬수도** 있습니다. 예를 들어, 아래와 같이 UserData 클래스를 변경하지 않고 UserData 클래스에서 MyData 클래스를 상속받아 작업한다면 UserData 클래스를 변경하지 않고도 작업이 가능합니다. 하지만 **UserData 클래스를 사용하는 개발자 입장에서 봤을 때는 분명이 생길 수 있는 작업**입니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

// UserData 클래스가 사용할 데이터의 최대 크기
#define MAX_USER_DATA_SIZE  8

class UserData
{
protected:
    // 값을 저장하기 위해 사용할 포인터
    // void *를 사용하는 것이 정석이지만 메모리 할당과 해제 그리고
    // 주소 연산시 char *를 사용하는 거이 더 편해서 char *로 설정함
    char *mp_data;
    // 저장된 값의 크기와 실제 할당된 크기
    int m_data_size, m_alloc_size;

public:
    UserData(int a_alloc_size = MAX_USER_DATA_SIZE)  // 생성자
    {
        mp_data = new char[a_alloc_size];
        m_data_size = 0;
        m_alloc_size = a_alloc_size;
    }

    ~UserData()  // 파괴자
    {
        // 할당된 메모리가 있다면 메모리를 해제한다.
        if (mp_data) delete[] mp_data;
    }

    // 외부에서 클래스 내부에 값을 저장할 때 사용하는 함수
    void SetData(void *ap_data, int a_data_size)
    {
        // 정상적인 정보인지 체크한다.
        if (a_data_size <= m_alloc_size) {
            // 할당된 메모리에 전달된 값을 저장한다.
            memcpy(mp_data, ap_data, a_data_size);
            // 전달된 크기를 저장한다.
            m_data_size = a_data_size;
        } else {
            // 정상적인 값이 전달되지 않았다면 복사된 크기 값을 초기화한다.
            m_data_size = 0;
        }
    }

    // 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
    void operator =(int a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }
```

```
    // 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
    void operator =(double a_data)
    {
        // 실숫값을 저장한다.
        SetData(&a_data, sizeof(double));
    }

    // 외부에서 클래스 내부에 저장된 값을 얻을 때 사용하는 함수
    void *GetData()
    {
        return mp_data;
    }

    // 내부적으로 가지고 있는 데이터의 크기를 얻을 때 사용하는 함수
    int GetDataSize()
    {
        return m_data_size;
    }

    // 정숫값을 반환하는 경우에 사용하는 함수
    operator int()
    {
        return *(int *)mp_data;
    }

    // 실숫값을 반환하는 경우에 사용하는 함수
    operator double()
    {
        return *(double *)mp_data;
    }
};

class MyData : public UserData
{
public:
    MyData(int a_alloc_size = MAX_USER_DATA_SIZE)  // 생성자
        : UserData(a_alloc_size)
    {

    }

    // 문자값을 반환하는 경우에 사용하는 함수
    operator char()
    {
        return *(char *)mp_data;
    }

    // 대입 연산자를 사용해서 문자값이 전달된 경우에 호출되는 함수
    void operator =(char a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(int));
    }
```

```
    }
};

int main()
{
    // 문자값, 정숫값 그리고 실숫값을 사용할 객체를 선언한다.
    MyData c_temp, i_temp, d_temp;

    // c_temp 객체에 'A' 문자를 저장한다.
    c_temp = 'A';
    // i_temp 객체에 5를 저장한다.
    i_temp = 5;
    // d_temp 객체에 6.2를 저장한다.
    d_temp = 6.2;

    // c_temp 객체, i_temp 객체 그리고 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "c_temp: " << (char)c_temp << endl;
    cout << "i_temp: " << (int)i_temp << endl;
    cout << "d_temp: " << (double)d_temp << endl;

    return 0;
}
```

2. 템플릿 클래스

C++ 언어는 **클래스를 정의할 때 멤버 변수의 자료형은 지정하지 않고 클래스를 사용하여 객체를 선언할 때 자료형은 지정하는 템플릿 클래스(template class) 문법을 추가로 제공**합니다.

예를 들어, 특정 **클래스가 사용하는 멤버 변수의 자료형은 int로 사용할 것인지 double로 사용할 것인지 결정할 수 없다면 아래와 같이 템플릿 형식으로 클래스를 선언**하면 됩니다. 이렇게 클래스를 정의하면 클래스가 사용할 멤버 변수(m_data)의 자료형을 지정하지 않고도 클래스를 정의할 수 있습니다.

```
template <class T> class MyData
{
private:
    T m_data;

public:
    void SetData(T a_data)
    {
        m_data = a_data;
    }

    T GetData()
    {
        return m_data;
    }
};
```

위와 같이 선언된 클래스는 멤버 변수의 자료형이 결정되지 않았기 때문에 **클래스를 사용해서 객체를 선언할 때 아래와 같이 멤버 변수의 자료형은 지정**해주면 됩니다.

```
MyData <int>i_temp;  // i_temp 객체의 m_data 멤버 변수는 int 자료형으로 처리된다.
```

위에 소개한 템플릿 클래스를 사용해서 1번의 마지막 예제를 변경해보면 다음과 같습니다.

```cpp
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

template <class T> class MyData
{
private:
    T m_data;

public:
    void SetData(T a_data)
    {
        m_data = a_data;
    }

    T GetData()
    {
        return m_data;
    }
};

int main()
{
    // 문자값과 실숫값을 사용할 객체를 선언한다.
    MyData <char>c_temp;
    MyData <int>i_temp;
    MyData <double>d_temp;

    // c_temp 객체에 'A' 문자를 저장한다.
    c_temp.SetData('A');
    // i_temp 객체에 5를 저장한다.
    i_temp.SetData(5);
    // d_temp 객체에 6.2를 저장한다.
    d_temp.SetData(6.2);

    // c_temp 객체, i_temp 객체 그리고 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "c_temp: " << c_temp.GetData() << endl;
    cout << "i_temp: " << i_temp.GetData() << endl;
    cout << "d_temp: " << d_temp.GetData() << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
c_temp: A
i_temp: 5
d_temp: 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

그리고 위 클래스에서 SetData 함수와 GetData 함수를 클래스 외부에 선언하는 형태로 사용하고 싶다면 아래와 같이 코드를 구성하면 됩니다.

```
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

template <class T> class MyData
{
private:
    T m_data;

public:
    void SetData(T a_data);
    T GetData();
};

template <class T> void MyData<T>::SetData(T a_data)
{
    m_data = a_data;
}

template <class T> T MyData<T>::GetData()
{
    return m_data;
}

int main()
{
    // 문자값과 실숫값을 사용할 객체를 선언한다.
    MyData <char>c_temp;
    MyData <int>i_temp;
    MyData <double>d_temp;

    // c_temp 객체에 'A' 문자를 저장한다.
    c_temp.SetData('A');
    // i_temp 객체에 5를 저장한다.
    i_temp.SetData(5);
    // d_temp 객체에 6.2를 저장한다.
    d_temp.SetData(6.2);

    // c_temp 객체, i_temp 객체 그리고 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "c_temp: " << c_temp.GetData() << endl;
    cout << "i_temp: " << i_temp.GetData() << endl;
    cout << "d_temp: " << d_temp.GetData() << endl;

    return 0;
}
```

3. 결론

어떤 클래스를 설계(정의)하면서 **사용할 데이터의 형식은 예측하기 어려운 경우, 나중에 어떤 자료형은 사용하게 되더라도 편하게 대처할 수 있는 코드른 만듣기 위해서는 클래스 구현 코드가 복잡해지기 마련**입니다. 따라서 무리하게 상황을 예측을 하거나 너무 다양한 대응 방법을 만드는 것보다는 사용할 자료형의 선택을 뒤로 미두는 것도 좋은 방법입니다.

하지만 이런 방법이 C 언어에서는 문법적으로 제공되지 않아 **#define 전처리기른 사용해서 매크로 형식으로 구조체나 함수른 정의하는 방법은 사용**했습니다. 예를 들어, 템플릿 클래스를 사용하지 않고 고전적인 #define 전처리기를 사용해서 템플릿 클래스 문법을 흉내내면 다음과 같습니다.

```
#include <iostream>      // 표준 입출력 스트림 객체를 사용하기 위해
using namespace std;     // std::를 생략하기 위해서

#define MyDataTemplate(T, class_name) \
class class_name \
{ \
private: \
    T m_data; \
\
public: \
    void SetData(T a_data)\
    {\
        m_data = a_data;\
    }\
\
    T GetData()\
    {\
        return m_data;\
    }\
}\

int main()
{
    // 문자값과 실숫값을 사용할 객체를 선언한다.
    MyDataTemplate(char, MyCharData) c_temp;
    MyDataTemplate(int, MyIntData)i_temp;
    MyDataTemplate(double, MyDoubleData) d_temp;

    // c_temp 객체에 'A' 문자를 저장한다.
    c_temp.SetData('A');
    // i_temp 객체에 5를 저장한다.
    i_temp.SetData(5);
    // d_temp 객체에 6.2를 저장한다.
    d_temp.SetData(6.2);

    // c_temp 객체, i_temp 객체 그리고 d_temp 객체가 가지고 있는 값을 출력한다.
    cout << "c_temp: " << c_temp.GetData() << endl;
    cout << "i_temp: " << i_temp.GetData() << endl;
    cout << "d_temp: " << d_temp.GetData() << endl;

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
c_temp: A
i_temp: 5
d_temp: 6.2

이 창을 닫으려면 아무 키나 누르세요...
```

하지만 C++ 언어는 템플릿 클래스라는 문법이 제공되어 #define은 사용하지 않고도 사용할 자료형의 선택은 실제로 객체를 선언하는 시점으로 미룬수 있습니다. 그래서 템플릿 클래스를 사용하면 전체적으로 코드가 간단해지기 때문에 클래스가 사용할 자료형을 예측하기 어려운 경우에는 템플릿 클래스 문법을 적절하게 활용하는 것이 좋습니다.

댓글 등록순 최신순

관심글 댓글 알림

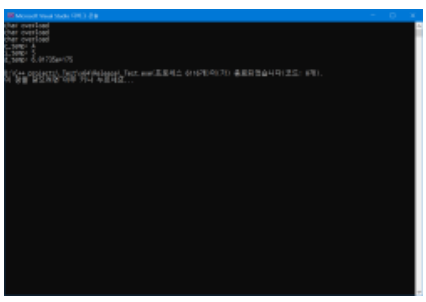
bac 2

질문 있습니다.
템플릿 클래스를 쓰지 않고 여러 자료형에 대응하는 클래스를 만드는 예제 코드 중 제일 마지막에 있는 상속 문법으로 사용할 자료형을 추가하는 코드에서 대입 연산자 오버로딩을 사용할 때 문자든 정수든 실수든 모두 자식 클래스가 갖고 있는 매개 변수가 char 자료형인 대입 연산자 오버로딩이 작동해버립니다.(컴파일러가 오해하는 현상을 방지하기 위해 int(5), double(6.2) 같은 형변환도 써봤지만 결과는 동일했습니다)

2022.08.22. 18:27 답글쓰기

bac 2

이 문제를 해결하기 위해서는 부모 클래스의 대입 연산자 오버로딩 함수를 명시적으로 호출해야 하나요? 그것 말고 다른 방법이 있다면 어떤 방법이 있을까요?



2022.08.22. 18:28 답글쓰기

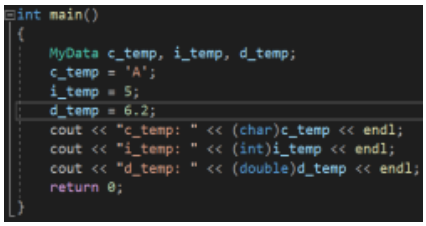
김성엽 M 작성자

bac main 함수는 어떻게 했나요?

2022.08.22. 18:30 답글쓰기

bac 2

김성엽 main() 함수는 본문과 동일하게 작성했습니다



```
int main()
{
    MyData c_temp, i_temp, d_temp;
    c_temp = 'A';
    i_temp = 5;
    d_temp = 6.2;
    cout << "c_temp: " << (char)c_temp << endl;
    cout << "i_temp: " << (int)i_temp << endl;
    cout << "d_temp: " << (double)d_temp << endl;
    return 0;
}
```

2022.08.22. 18:33 답글쓰기

김성엽 M 작성자

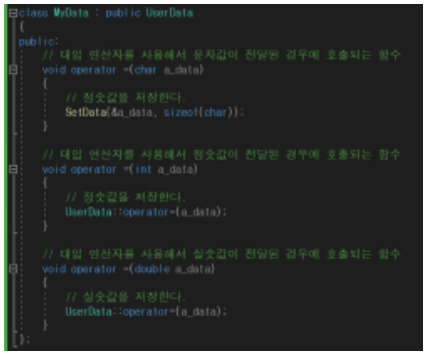
bac 형변환 연산자 오버로딩과 달리 대입 연산자 오버로딩은 부모 자식간에 직접 사용이 불가능합니다. 따라서 MyData에서 부모에 사용한 대입 연산자 오버로딩을 사용하려면 아래와 같이 함수를 추가해서 사용해야 합니다.

```
class MyData : public UserData
{
public:
// 대입 연산자를 사용해서 문자값이 전달된 경우에 호출되는 함수
void operator =(char a_data)
{
// 정숫값을 저장한다.
SetData(&a_data, sizeof(char));
}

// 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
void operator =(int a_data)
{
// 정숫값을 저장한다.
UserData::operator=(a_data);
}

// 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
void operator =(double a_data)
{
// 실숫값을 저장한다.
UserData::operator=(a_data);
}

};
```

2022.08.22. 20:02 답글쓰기

김성엽 M 작성자

bac

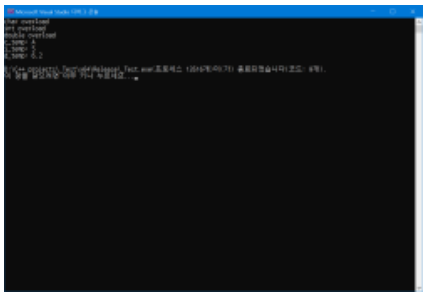


```
class MyData : public UserData
{
public:
    // 대입 연산자를 사용해서 문자값이 전달된 경우에 호출되는 함수
    void operator =(char a_data)
    {
        // 정숫값을 저장한다.
        SetData(&a_data, sizeof(char));
    }

    // 대입 연산자를 사용해서 정숫값이 전달된 경우에 호출되는 함수
    void operator =(int a_data)
    {
        // 정숫값을 저장한다.
        UserData::operator=(a_data);
    }

    // 대입 연산자를 사용해서 실숫값이 전달된 경우에 호출되는 함수
    void operator =(double a_data)
    {
        // 실숫값을 저장한다.
        UserData::operator=(a_data);
    }
};
```

2022.08.22. 20:04 답글쓰기

bac 2

김성엽 이제는 결과가 정상적으로 나오네요. 접근 지정자로 제한된 게 아닌데도 자식 클래스에서 직접 사용할 수 없는 부모 클래스 함수도 있다는 사실 하나 알게 됐네요.



2022.08.22. 20:17 답글쓰기

김성엽 M 작성자

bac ㅎㅎ 파이팅입니다~ :)

2022.08.22. 21:03 답글쓰기

해피파파 3

질문) 메모리 재 할당률을 줄이고 싶다면 -> 무슨 뜻인지요? 왜 MAX_USER_DATA_SIZE를 설정하는지 모르겠습니다.

2022.11.10. 10:10 답글쓰기

김성엽 M 작성자

메모리를 매번 할당했다가 해제하지않고 공통으로 사용할수 있는 크기를 미리 생성해놓고 사용하는 방식을 소개한것입니다

2022.11.10. 10:12 답글쓰기

김성엽 M 작성자

메모리 사용량을 예상할수 있다면 저렇게 하면 객체가 만들어져서 제거 될때까지 계속 사용하는 것이 더 좋습니다

2022.11.10. 10:14 답글쓰기

해피파파 3

c언어는 struct만 지원하지, class문법은 지원하지 않지요?
#define으로 탬플릿클래스를 흉내냈다고 하는 것은 C++에서 이야기지, C에서는 해당사항이 없는거지요? (개념이 혼동되서^^;)

2022.11.10. 10:17 답글쓰기

김성엽 M 작성자

네~ class문법은 c++만 사용가능합니다~ :)

2022.11.10. 10:18 답글쓰기

카일 3

다형성 등 이해가 부족한 상태에서 템플릿 클래스를 남용하지 말아야겠다는 생각이 들었습니다. 강의 잘 들었습니다.

2022.11.26. 19:03 답글쓰기

김성엽 M 작성자

그렇습니다. 템플릿보다는 다형성을 더 많이 사용하면서 구조 설계에 익숙해 지는것이 좋습니다

2022.11.26. 19:07 답글쓰기

카일 3

김성엽 넵~ 잘 기억하겠습니다

2022.11.26. 19:09 답글쓰기

김성엽 M 작성자

카일

2022.11.28. 23:04 답글쓰기

티모 3

구조 설계부터 익숙해져야겠어요

2022.11.28. 21:25 답글쓰기

dh221009

댓글을 남겨보세요

등록