싱글톤(Singleton) 패턴

- 학습지식 개요/요점
  - 싱글톤 패턴이란
  - 싱글톤 패턴의 설계
  - 싱글톤 패턴 예제 작성

❑ 싱글톤 패턴이란

- 싱글톤 패턴이란 특정 클래스의 인스턴스를 1개만 생성하도록 제한하고, 이를 공유하는 방법이다.
- 생성자를 private으로 하고, 자기 자신을 받는 함수를 만들어서 자기 자신의 객체를 반환하는 클래스이다.

❑ 싱글톤 패턴의 필요성

- 프로그램 내에 특정 클래스 내에서 하나의 객체만 생성하고자 할 때 사용한다.
- 환경 설정 객체와 같이 공통으로 사용해야 하는 모듈에서 사용한다.

❑ 일반적인 싱글톤 패턴의 설계 (정적 싱글톤)

```
class SingletonClass
{
private:
        static SingletonClass instance;
        SingletonClass(){}
        ~SingletonClass(){}
public:
        static SingletonClass& getInstance()
        {
                return instance;
        }
};

SingletonClass SingletonClass::instance;
```

- 생성자가 private인 이유는 외부에서 객체 생성을 막기 위함이다.
- 어디서든 싱글톤 객체를 접근할 수 있게 하기 위해서 static 함수를 정의하였다.

실습 예제 및 수행가이드

- **싱글톤 패턴 예제 작성**

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

class SingletonClass
{
private:
		static SingletonClass instance;
		SingletonClass(){}
		~SingletonClass(){}
public:
		static SingletonClass& getInstance(){
			return instance;
		}
		void ShowMessage(){
			cout<<"Singleton Pattern"<<endl;
		}
};
SingletonClass SingletonClass::instance;

void main()
{
		SingletonClass::getInstance().ShowMessage();
}
```

❑ 일반적인 싱글톤 패턴의 설계 (정적 싱글톤)

- **정적 싱글톤의 문제점**

  - static 클래스 멤버변수는 프로그램 시작 시 초기화되므로 해당 클래스 사용 여부와 상관 없이 생성

  - 다른 전역 객체의 생성자에서 해당 멤버를 참조하고 싶을 때 문제가 발생할 수도 있다. 예를 들어 싱글톤 클래스의 instance가 생성 되기 전에 다른 전역 객체에서 instance를 참조하게 되면 문제가 발생한다.

  - C++은 전역 객체들의 생성 순서에 대해서는 명확하게 정의되어 있지 않기 때문이다.

```
SingletonClass::getInstance().ShowMessage();
```

- **해결 방법**

  - 객체의 생성 시점을 조절.

  - 늦은 초기화 기법을 사용, 즉 인스턴스를 얻어오는 함수 내에서 인스턴스를 생성.

❑ 동적인 싱글톤 패턴의 설계

```
class SingletonClass
{
private:
        static SingletonClass* instance;
        SingletonClass(){}
public:
        ~SingletonClass(){}
        static SingletonClass *getInstance()
        {
                if(instance == NULL)
                        instance = new  SingletonClass();
                return instance;
        }
};
```

- 함수 내에서 new 연산자로 static SingletonClass 객체를 생성
- instance의 생성 여부를 판단하는 NULL 체크

실습 예제 및 수행가이드

- **동적 싱글톤 패턴 예제 작성**

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

class SingletonClass
{
private:
        static SingletonClass* instance;
        SingletonClass(){}
public:
        ~SingletonClass(){}
        static SingletonClass *getInstance()
        {
                if(!instance)
                        instance = new SingletonClass();
                return instance;
        }
        void ShowMessage()
        {
                cout<<"Singleton Pattern"<<endl;
        }
};
```

실습 예제 및 수행가이드

- **동적 싱글톤 패턴 예제 작성**

```
SingletonClass* SingletonClass::instance = NULL;

void main()
{
        SingletonClass::getInstance()->ShowMessage();
}
```

실습 예제 및 수행가이드

- **싱글톤 패턴 예제 작성2**

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

class SingletonClass
{
private:
        static SingletonClass* instance;
        int total;
        SingletonClass();
public:
        ~SingletonClass(){}
        static SingletonClass *getInstance()
        {
                if(instance == NULL)
                        instance = new SingletonClass();
                return instance;
        }
        void addValue(int value);
        int getTotalValue();
};
```

실습 예제 및 수행가이드

- **싱글톤 패턴 예제 작성2**

```
SingletonClass* SingletonClass::instance = NULL;

SingletonClass::SingletonClass()
{
        total = 0;
}
void SingletonClass::addValue(int value)
{
        total = total + value;
}
int SingletonClass::getTotalValue()
{
        return total;
}
```

실습 예제 및 수행가이드

- **싱글톤 패턴 예제 작성2**

```
void main()
{
        SingletonClass* ins1 = SingletonClass::getInstance();
        SingletonClass* ins2 = SingletonClass::getInstance();
        SingletonClass* ins3 = SingletonClass::getInstance();

        ins1->addValue(10);
        cout<<"total : "<<ins1->getTotalValue()<<endl;
        ins2->addValue(10);
        cout<<"total : "<<ins1->getTotalValue()<<endl;
        ins3->addValue(10);
        cout<<"total : "<<ins1->getTotalValue()<<endl;
}
```

- **멤버변수 total이 누적되는 것을 확인할 수 있다.**
- **결국 ins1, ins2, ins3 모두 같은 객체임을 확인할 수 있다.**

❑ 동적 싱글톤 패턴의 문제점

- new 연산자를 통해 동적으로 생성한 객체에 대해 소멸을 보장받지 못한다.
- 그로 인해 메모리 누수가 생긴다.
- 이 문제를 해결하기 위해 프로그램 종료 직전 소멸 루틴을 직접 호출해 주도록 한다.

실습 예제 및 수행가이드

- **싱글톤 패턴 소멸 루틴**

```
#include<iostream>
#include <crtdbg.h>
using namespace std;

class SingletonClass
{
private:
        static SingletonClass* instance;
        SingletonClass(){}
        ~SingletonClass(){}
public:
        static SingletonClass *getInstance()
        {
                if(!instance)
                        instance = new SingletonClass();
                return instance;
        }
        void ShowMessage()
        {
                cout<<"Singleton Pattern"<<endl;
        }
```

실습 예제 및 수행가이드

- **싱글톤 패턴 소멸 루틴**

```
        void destroy()
        {
                if(instance!=NULL)
                {
                        delete instance;
                        instance = NULL;
                        cout<<"instance 해제"<<endl;
                }
        }
};

SingletonClass* SingletonClass::instance = NULL;

void main()
{
        _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
        SingletonClass::getInstance()->ShowMessage();
        SingletonClass::getInstance()->destroy();
}
```