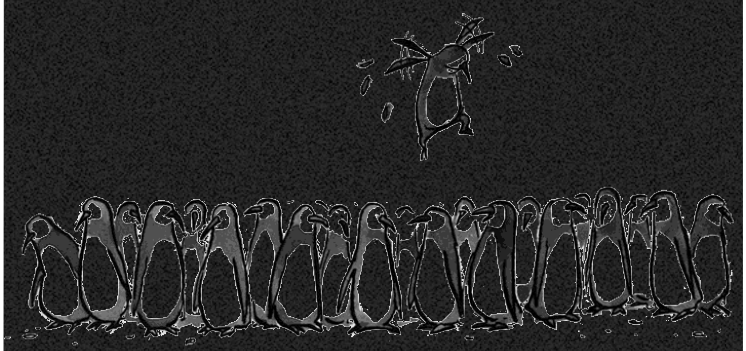

디자인 패턴

C++

윤찬식 | 아이오교육센터

www.ioacademy.co.kr

싱글톤을 만드는 기술

이 장에서는 C++ 에서 싱글톤을 만드는 기술을 정리해봅시다.

싱글톤은 GoF 의 디자인 패턴중 가장 유명한 패턴입니다.
"클래스 인스턴스는 오직 하나임을 보장하며
이에 대한 접근은 어디에서든지 하나로만 통일하여 제공한다."

C++ 에서 싱글톤을 만드는 가장 일반적인 방법은 Effective C++ 에서 스콧 마이어스에 의해 언급된 정적 멤버 함수 내부에 객체를 만드는 것 입니다. 그의 이름을 따서 마이어스의 싱글톤(Meyers's Singleton) 이라고 부릅니다.

Cursor 라는 객체를 프로그램 상에서 오직 한개만을 생성하고 싶다면 아래와 같이 구현 가능합니다.

```
class Cursor {
 private:
  Cursor() {}

  Cursor(const Cursor&);
  void operator=(const Cursor&);

 public:
  static Cursor& getInstance() {
    static Cursor instance;
    return instance;
  }
};
```

싱글톤을 만드는 방법은 규칙 3가지가 중요합니다.
첫번째 규칙. private 생성자
두번째 규칙. 오직 한개의 인스턴스의 참조만을 리턴하는 정적 멤버 함수
세번째 규칙. 복사와 대입 금지

첫번째 규칙인 private 생성자는 클래스 외부에서 혹시 모를 객체의 생성을 방지하고, 두번째 규칙은 내부 정적 객체로 생성된 유일한 인스턴스에 접근하는 인터페이스를 제공합니다. 마지막 규칙은 C++ 에서 자동적으로 생성되는 복사 생성자와 대입 연산자의 선언만을 두는 복사 금지 기법을 통해 클래스 외부에서 객체의 복사를 방지하며, 클래스 내부에서 혹시 모를 복사와 대입을 방지하기 위해 선언만 두는 형태로 구현하는 것입니다. 만약 클래스 내부에서 문제가 발생한다면 링킹 에러를 통해 싱글톤이 깨지는 것을 방지할 수 있습니다.

C++ 에서 복사와 대입을 선언만을 통해 방지하는 기법은 C++11 에서는 delete function 이라는 문법을 통해 구현할 수 있습니다.

```
class Cursor {
 private:
  Cursor() {}

  Cursor(const Cursor&) = delete;
  void operator=(const Cursor&) = delete;

 public:
  static Cursor& getInstance() {
    static Cursor instance;
    return instance;
  }
};
```

delete function 을 통해 자동적으로 생성하는 복사 생성자와 대입 연산자를 삭제할 수 있으며, 클래스 외부에서의 접근 뿐 아니라 클래스 내부에서 발생되는 복사 및 대입 연산자의 접근을 컴파일 에러를 통해서 검출할 수 있습니다.

위의 싱글톤을 생성하는 규칙과 복사 금지의 코드를 구현하는 방법은 항상 동일하기 때문에, 매크로를 통해서 일반화한다면 또 다른 싱글톤 또는 복사 금지가 필요할 때 유용하게 사용할 수 있습니다.

```
#define MAKE_NOCOPY(classname)           \
 private:                                \
  classname(const classname&) = delete; \
  void operator=(const classname&) = delete;

#define MAKE_SINGLETON(classname)    \
 private:                            \
  MAKE_NOCOPY(classname)             \
  classname() {}                     \
                                     \
 public:                             \
  static classname& getInstance() {  \
    static classname instance;       \
    return instance;                 \
  }

class Mouse {
  MAKE_SINGLETON(Mouse)
};
```

마이어스의 싱글톤은 C++11/14 부터는 static 정적 객체의 초기화 과정에 대한 스레드 안정성 까지 보장되기 때문에 안전하게 사용할 수 있습니다.

> If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.
>
> §6.7 [stmt.dcl]

또한 GCC 컴파일러는 별도의 컴파일 옵션(-fthreadsafe-statics)을 통해 함수 내부의 정적 객체에 대한 스레드 안정성 보장을 위한 기능도 제공하고 있습니다. 하지만 C++98/03 표준에서는 스레드 안전성에 대한 언급이 존재하지 않기 때문에 스레드 안전성에 대한 추가적인 고려가 필요합니다.

싱글톤의 생성 시점에 스레드 안전성을 보장하기 위해서 가장 쉬운 접근 방법은 API design for C++에서 언급된 정적 초기화 방법이나 명시적 API 초기화 방법을 사용하는 것도 좋은 방법이 될 수 있습니다.

```
// main이 처음 호출되었을 시점에는 단일 스레드 이므로 안전하다.
int main() {
  Cursor::getInstance();
}
```

많은 C++ 기반의 오픈 소스도 마이어스의 싱글톤을 사용하고 있습니다.

```
CSSValuePool& CSSValuePool::singleton()
{
    static NeverDestroyed<CSSValuePool> pool;
    return pool;
}
```

Webkit(./Source/WebCore/css/CSSValuePool.cpp)

```
/* static */
OpenSLESProvider& OpenSLESProvider::getInstance()
{
  static OpenSLESProvider instance;
  return instance;
}
```

Firefox(./dom/media/systemservices/OpenSLESProvider.cpp)

Cursor 싱글톤 인스턴스를 힙에 생성하기 위해서는 다음과 같이 구현할 수 있습니다.

```
class Cursor {
 private:
  Cursor() {}
  Cursor(const Cursor&);
  void operator=(const Cursor&);

  static Cursor* sInstance;

 public:
  static Cursor& getInstance() {
    if (sInstance == 0) sInstance = new Cursor;
    return *sInstance;
  }
};

// static 멤버 변수는 반드시 외부에 선언을 해야 합니다.
Cursor* Cursor::sInstance = 0;

int main() { Cursor& c1 = Cursor::getInstance(); }
```

하지만 위의 싱글톤은 멀티 스레드에서 getInstance() 가 호출될 경우 인스턴스가 여러개 생성되는 문제점이 있습니다. 따라서 Mutex 를 통해 sInstance 의 인스턴스를 체크하고 new 를 통한 객체 생성을 보호해주어야 합니다.

```
class Mutex {
 public:
  void lock() { cout << "Mutex Lock" << endl; }
  void unlock() { cout << "Mutex Unlock" << endl; }
};
```

```
class Cursor {
 private:
  static Cursor* sInstance;
  static Mutex sLock;

 public:
  static Cursor& getInstance() {
    sLock.lock();
    if (sInstance == 0) sInstance = new Cursor;
    sLock.unlock();
    return *sInstance;
  }
};

Mutex Cursor::sLock;
Cursor* Cursor::sInstance = 0;
```

Mutex 를 통해 싱글톤의 객체 생성 시점을 보호해준다면, 멀티 스레드 상에서 객체가 여러개가 생성되는 문제를 방지할 수 있습니다. 하지만 위처럼 뮤텍스를 lock 과 unlock 의 명시적 호출의 형태로

사용한다면 예외가 발생할 경우 데드락의 위험이 있습니다. Cursor 의 인스턴스를 할당하는 new 연산은 가용한 메모리가 없다면 std::bad_alloc 이라는 예외가 발생하게 됩니다. 예외가 발생하게 되면 뮤텍스의 잠금을 푸는 코드가 실행되지 않고 getInstance() 를 빠져나가게 됩니다. 만약 다른 스레드에 의해서 getInstance 가 호출된다면 뮤텍스의 잠금은 영원히 해지할 수 없기 때문에 데드락에 빠지게 될 것입니다.

C++ 에서는 뮤텍스 같은 사용 후 꼭 해지되어야 하는 자원에 대해서는 명시적인 잠금과 해지를 사용하기보다는 RAII(Resource Acquisition Is Initialization) 을 통해 사용하는 것이 일반적입니다.

```
class Mutex {
 public:
  void lock() { cout << "Mutex lock" << endl; }
  void unlock() { cout << "Mutex unlock" << endl; }

  class Autolock {
    Mutex& mLock;

   public:
    Autolock(Mutex& m) : mLock(m) { mLock.lock(); }
    ~Autolock() { mLock.unlock(); }
  };
};
```

```
static Cursor& getInstance() {
  Mutex::Autolock al(sLock);
  if (sInstance == 0) sInstance = new Cursor;

  return *sInstance;
}
```

위의 방법을 통해 뮤텍스를 사용한다면, al 이라는 지역 객체의 생성자를 통해 lock 이 호출되고 소멸자를 통해 unlock 이 자동적으로 호출됩니다. 또한 객체를 초기화하는 중 예외가 발생한다고 해도 C++ 표준에서는 예외 발생으로 인해 함수를 빠져나갈 때, 지역 객체에 대한 소멸자 호출을 보장하고 있기 때문에 al의 소멸자는 정상적으로 호출될 것 입니다. 따라서 데드락은 발생하지 않을 것입니다. 이처럼 생성자와 소멸자를 통해 자원을 관리하는 RAII 는 자원 해지를 편리하게 사용할 수 있을 뿐 아니라 예외가 발생하였을 경우 자원 누수 및 데드락을 방지하는데 유용하게 사용됩니다.

싱글톤의 getInstance 를 뮤텍스를 통해 동기화하면 멀티 스레드에 의해 객체가 여러개 생성되는 것을 방지할 수는 있지만, 싱글톤 인스턴스에 접근하기 위해서는 결국 뮤텍스 동기화가 필요하므로 성능적으로 저하가 있습니다. 그래서 생성 이후에 불필요한 뮤텍스 동기화를 방지하기 위해 사용하는 방법이 있습니다.

```
static Cursor& getInstance() {
  if (sInstance == 0) {
    Mutex::Autolock al(sLock);
    if (sInstance == 0) sInstance = new Cursor;
  }
  return *sInstance;
}
```

위의 방법은 흔히 Double Checked Locking Pattern(DCLP) 로 알려져 있습니다. 하지만 위의 방법은 멀티 프로세서 머신에서 CPU의 명령어 비순차 처리에 의해 제대로 동작하지 않습니다.
(참고 : C++ and the Perils of Double-Checked Locking, 2004)
결국 DCLP를 제대로 동작하게 위한 방법은 C++98/03 에서는 제공되지 않기 때문에, 절대 사용하면 안됩니다. 다행스럽게도 C++11/14 에서는 메모리 장벽에 대한 기능이 제공되기 때문에 DCLP 를 안전하게 구현 가능하지만, 아래의 방법을 통해 힙에 생성하는 싱글톤을 좀 더 쉽게 구현할 수 있습니다.

```
static Cursor& getInstance() {
  static Cursor* instance = new Cursor;
  return *instance;
}
```

C++11/14 Heap Singleton

안드로이드 프레임워크는 CRTP(Curiosly Recurring Template Pattern)을 통해 힙에 생성하는 싱글톤을 일반화하여 사용하고 있습니다.

```
template <typename TYPE>
class ANDROID_API Singleton
{
public:
    static TYPE& getInstance() {
        Mutex::Autolock _l(sLock);
        TYPE* instance = sInstance;
        if (instance == 0) {
            instance = new TYPE();
            sInstance = instance;
        }
        return *instance;
    }

    static bool hasInstance() {
        Mutex::Autolock _l(sLock);
        return sInstance != 0;
    }

protected:
    ~Singleton() { };
    Singleton() { };

private:
    Singleton(const Singleton&);
    Singleton& operator = (const Singleton&);
    static Mutex sLock;
    static TYPE* sInstance;
};
```

AOSP Singleton

```
class Cursor : public Singleton<Cursor> {

};
```

싱글톤은 프로그램 상에서 하나만 존재하지 않는 자원을 표현할 때 유용하게 사용되는 패턴입니다. 또한 여러 리소스에 접근하기 위한 하나의 포인트를 제공하는 매니저 클래스를 구현할 때도 유용하게 사용됩니다. 하지만 싱글톤은 편리함을 제공하는 대신 클래스 간의 결합도를 높이기 때문에, 이후에 프로그램을 리팩토링 하거나 격리된 부분을 테스트 하기 위한 단위 테스트 코드 작성도 어렵게 만듭니다.

핵심

- 싱글톤을 만드는 규칙 3가지
- 힙에 만드는 싱글톤
- 스레드 안전성에 대한 고려
- C++11/14 싱글톤

참고 자료

- Effective C++ 항목4: 객체를 사용하기 전에 반드시 그 객체를 초기화하자.
- API design for C++ 3.2 싱글톤
- C++ 표준 문서 §6.7 [stmt.dcl]
- http://gameprogrammingpatterns.com/singleton.html
- http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/