항목 29 | 예외 안전성이 확보되는 그날 위해 싸우고 또 싸우자!

예외 안전성을 가진 함수라면 예외가 발생할 때 이렇게 동작해야 합니다.

* **자원이 새도록 만들지 않습니다.**
* **자료구조가 더럽혀지는 것을 허용하지 않습니다.**

위 동작과 완전 반대가 되어 예외 안전성이라는 측면에서 볼 때 “이보다 더 나쁠 수 없다.” 라고 말 할 수 있는 코드를 보여 드리고 설명 하겠습니다.

*class PrettyMenu {*

*public:*

*void changeBackground(std::istream& imgSrc); // 배경그림을 바꾸는 함수*

*private:*

*Mutex mutex; // 이 객체 하나를 위한 mutex*

*Image\* bgImage; // 현재의 배경그림*

*int imageChanges; // 배경그림이 바뀐 횟수*

*};*

*// 함수 구현부*

*void PrettyMenu::changeBackground(std::istream& imgSrc)*

*{*

*lock(&mutex); // mutex 를 획득합니다*

*delete bgImage; // 이전의 배경그림을 없앱니다.*

*++imageChanges; // 그림 변경 횟수를 갱신합니다.*

*bgImage = new Image(imgSrc); // 새 배경그림을 깔아 놓습니다.*

*unlock(&mutex); // mutex 를 해제합니다.*

*}*

위 코드에서 자원이 샐 수 있는 부분이 존재 합니다. changeBackground함수의 new Image(imgSrc)에서 예외가 발생하면 unlock(&mutex) 코드를 실행 시킬 수 없어 mutex가 계속 잡힌 상태로 남기 때문 입니다.

또한 자료구조가 더럽혀질 수 있는 부분이 있습니다. 이번에도 함수의 new Image(imgSrc)에서 예외가 발생하면 bgImage는 이미 delete되어 있는 상태인데 새 배경그림을 얻지 못 한데다 imageChanges(그림 변경 횟수)도 증가 된 상태입니다.

자원 누출 문제는 자원 관리 객체를 사용하면 쉽게 해결 할 수 있습니다.

자료구조 오염 문제를 해결하기 위해 3가지 예외 안전성 보장 중 하나를 선택해야 합니다.

* **기본적인 보장**

**예외가 발생하면, 모든 객체, 변수 등 값을 유효한 값으로 유지하겠다는 보장 입니다.** 객체, 자료구조 등 더럽혀지지 않습니다. 다만 예외 발생 후 객체의 상태는 알 수 없습니다. 예를 들어 changeBackground함수가 기본적인 보장을 제공하게 만들어져 있고 이 때 예외가 발생하면 PrettyMenu 객체는 이전 배경그림을 계속 그릴 수 도 있고 처음 만들어 둔 배경을 그릴 수 도 있습니다.

* **강력한 보장**

**예외가 발생하면, 프로그램의 상태를 절대로 변경하지 않겠다는 보장 입니다.** 함수 호출 중에 문제가 발생하면 함수 호출 이전으로 되돌아갑니다.

* **예외불가 보장**

**예외를 절대로 던지지 않겠다는 보장입니다.** 약속한 동작은 언제나 완수하는 함수 입니다. 오해 할 수 있는 부분은 예외를 던지지 않게 지정 된 함수는 예외불가 보장을 제공한다는 부분 입니다. 하지만 이는 잘 못 된 생각입니다. 예외를 던지지 않을 뿐, 예외가 발생하면 어떠한 처리도 하지않을 수 있습니다.

모든 함수는 위 세 가지 보장 중 하나를 반드시 제공 해야 합니다. 어떤 보장을 제공할지 고르는 것은 프로그래머 몫 입니다.

예외불가 보장이 가장 훌륭하겠지만 c++특성상 예외를 던지지 않기란 힘듭니다. 그래서 현실적으로 대부분의 함수는 기본적인 보장과 강력한 보장 중 하나를 고르게 됩니다.

이제 changeBackground함수를 기본적인 보장으로 만들어 보겠습니다.

*class PrettyMenu {*

*shared\_ptr<Image> bgImage; // 자원관리 포인터로 바꿈*

*};*

*void PrettyMenu::changeBackground ( istream& imgSrc )*

*{*

*Lock m1 ( &mutex );*

*bgImage.reset ( new Image ( imgSrc ) ); // bgImage의 내부 포인터를 new Image( ) 실행결과로 바꿔치기 합니다.*

*++imageChanges;*

*}*

**이제 bgImage를 직접 삭제할 필요가 없습니다. new Image ( imgSrc )가 성공적으로 실행 되면 알아서 삭제가 될 것입니다. 그리고 배경이 성공적으로 바뀌어야 마지막에 imageChanges를 증가 시키기 때문에 객체의 상태가 오염되지 않습니다.**

이번에는 changeBackground함수를 강력한 보장으로 만들어 보겠습니다.

간단하게 원리를 살펴보면 강력한 보장은 복사 후 맞바꾸기를 실행 하는 것 입니다. 어떤 객체를 수정하기 전에 미리 사본을 복사해 두고 사본을 수정 합니다. **성공적으로 수정이 완료 되면 원본과 바꾸는 작업을 합니다.**

객체의 모든 데이터를 별도의 struct 또는 class에 넣어두고, 넣어둔 객체의 포인터를 객체가 가지고 있게 만듭니다.

*// 모든 데이터를 struct에 넣어둠.*

*struct PMImpl {*

*shared\_ptr<Image> bgImage;*

*int imageChanges;*

*};*

*// 원본 객체*

*class PrettyMenu {*

*private:*

*Mutex mutex;*

*shared\_ptr<PMImpl> pImpl; // 넣어둔 객체의 포인터를 가지고 있음.*

*};*

*// 함수*

*void PrettyMenu::changeBackground(istream& imgSrc)*

*{*

*using std::swap; // 항목 25 참고 하세요.*

*Lock m1 ( &mutex ) // mutex를 잡고 들어갑니다.*

*shared\_ptr<PMImpl> pNew ( new PMImpl ( \*pImpl ) ); // 사본을 만들기 위해 원본 객체의 데이터 부분을 복사*

*pNew -> bgImage.reset( new Image ( imgSrc ) ); // 사본을 수정 합니다.*

*++(pNew -> imageChanges); // 사본을 수정 합니다.*

*swap ( pImpl, pNew ); // 원본을 사본으로 대체*

*unlock ( &mutex ) // mutex 해제*

*}*

하지만 대부분의 함수들은 강력한 예외 안전성을 보장하지 않습니다. 이유는 다음 함수를 예를 들어 설명 하겠습니다.

*// 복사 후 맞바꾸기 기능이 들어간 강력한 에외 안전성을 가진 함수 입니다.*

*void someFunc ( )*

*{*

*. . . // 사본 생성 등.*

*f1( ); // 강력한 예외 안전성을 보장하지 않는 함수를 호출(기본적인 보장)*

*f2( );*

*. . . // 변경된 상태를 바꾸어 넣습니다.*

*}*

**f1 함수 호출로 인해 사본에 저장하지 않은 다른 객체의 상태가 바뀔 수 도 있고, f2가 만약 데이터베이스를 변경하는 함수라면 someFunc ( )함수는 손을 쓸 수 없습니다.**

강력한 예외 안전성을 보장하지 않는 또 다른 이유는 효율 문제도 있습니다. 함수를 호출 할 때마다 복사 후 맞바꾸기 작업을 진행합니다. 복사에 걸리는 시간을 감수 해야 합니다. 따라서 실용성이 확보될 때만 강력한 보장을 제공하는 것이 좋습니다.

**예외에 반만 안전하다 라는 말은 없습니다. 안전 하거나 불안전 하거나 둘중 하나 입니다.** “어떻게 하면 예외에 안전한 코드를 만들까” 고민하는 버릇을 들이세요.