Chapter3. 가상 함수

**가상 함수 역학**

- 만약 가상 함수를 사용하고 싶지 않다면, **형식에 대한 확인 코드를** 제공해 동적 바인딩을 회피할 수 있다.

Ex)

- ZooAnimal 기본 클래스

- 나머지 동물들은 ZooAnimal 로부터 파생된다.

|  |
| --- |
| class ZooAnimal  {  public:  virtual void draw();  int resolveType() { return myType; }  private:  int myType;  };  class Bear : public ZooAnimal  {  public:  void draw();  }; |

- 만약 동물원의 모든 동물을 그리고 싶다면 다음과 같은 **형식에 대한 확인을** 해야함

|  |
| --- |
| void drawAllAnimals()  {  for (first animal->last animal)  {  switch (animal->resolveType())  {  case BEAR:  // Bear, draw 호출  break;  case MONKEY:  // Monkey, draw 호출  break;  }  }  } |

- 이 코드는 유지, 보수가 까다롭다. 동물이 동물원을 떠날 때 마다 형식 확인을 위한 switch 문을 수행해야 하기 때문

- 가상 함수의 동적 바인딩을 사용하면 코드의 이러한 형식 확인에 대한 의존성을 없앨 수 있다.

|  |
| --- |
| void drawAllAnimals()  {  for (first animal->last animal)  animal->draw();  } |

- 위 코드 또한 Monkey와 Bear를 구별할 수 있다.

가상 함수는 성능상의 부하를 몇 가지 방식으로 부하하는 것처럼 보일 수 있다.

1. vptr이 생성자에서 초기화 되어야 한다.

2. 가상 함수는 포인터 간접 참조를 통해 호출, 함수 테이블에 대한 포인터를 반입해 온 다음 올바른 함수에 접근해야 한다.

3. 인라인은 컴파일 시에 결정된다. 컴파일러는 실행 시에 확인되는 가상 함수를 인라인으로 만들 수 없다.

C++에 공정하기 위해, 처음 두 항목을 인과 성능 부하라고 여기면 안될 것이다.

1. 부하는 Bear 구현에서 형식 멤버를 초기화 하는 것과 동일하다.

|  |
| --- |
| class Bear : public ZooAnimal  {  public:  void draw();  }; |

2. 부하는 Bear::draw()를 Monkey::draw()와 구별하기 위해 필요한 switch 구문 논리와 동일하다.

|  |
| --- |
| switch (animal->resolveType())  {  case BEAR:  // Bear, draw 호출  break;  case MONKEY:  // Monkey, draw 호출  break;  } |

그렇다면 가상 함수의 진정한 부하는 3. 항목뿐이다. 가상 함수를 인라인으로 만들 수 없다는 것은 커다란 부하 일 수 있다.

- 대부분의 가상 함수 호출에서, 확인은 오직 실행 시에 일어날 수 밖에 없다. 왜냐하면 함수가 호출될 객체의 형식은 컴파일 시에 확인될 수 없기 때문이다.

- 인라인은 특정 함수에 대한 지식을 필요로 하는 컴파일 시의 결정이다. 컴파일시에 어떤 함수를 호출할지 결정할 수 없다면 함수를 인라인으로 만들 수 없다.

**템플릿과 상속**

- 가상 함수 호출은 인라인일 될 수 없다. 이것은 성능상의 문제를 일으킬 수 있다. 함수 호출의 동적 바인딩은 상속의 결과물이다.

- 동적 바인딩을 제거할 수 있는 한 가지 방법은 상속을 템플릿 기반 디자인으로 교체하는 것이다.

- 템플릿은 실행시가 아닌, 컴파일 시에 확인 되기 때문에 좀더 성능 친화적이라 할 수 있다.

Ex) win32 환경에서 동시에 여러 스레드가 안전하게 조작할 수 잇는 스레드 안전한 문자열 클래스를 개발

- 동기화 모델중 하나를 선택할 수 있도록 상속 구조의 Locker를 구현한다.

|  |
| --- |
| class Locker {  public:  Locker() {}  virtual ~Locker() {}  virtual void lock() = 0;  virtual void unlock() = 0;  };  class CriticalSectionLock : public Locker {};  class MutexLock : public Locker {}; |

- 스레드 안전한 문자열은 기존 표준 string으로부터 파생하기로 결정

- 나머지 디자인 결정 사항은 다음과 같다.

**1. 하드 코딩 –** 세가지 다른 클래스를 string으로부터 파생시킬 수 있다.(CriticalSectionString, MutexString, 등) 각 클래스는 연관된 동기화 메커니즘을 구현하고 있다.

**2. 상속 –** Locker 객체에 대한 포인터를 포함하는 ThreadSafeString 클래스 하나를 파생시킬 수도 있다. 다형성을 사용하여 특정 동기화 메커니즘을 실행 시에 선택한다.

**3. 템플릿 –** Locker 형식을 매개변수로 받아들이는 템플릿 기반 문자열 클래스를 만든다.

**하드코딩**

표준 string 클래스로부터 파생된 각 클래스는 특정 동기화 메커니즘을 수행

동기화 모델 별로 string 객체를 선언해야 한다.

|  |
| --- |
| class CriticalSectionString : public std::string {  public:  int length()  {  cs.lock();  int len = std::string::length();  cs.unlock();  return len;  }  private:  CriticalSectionLock cs;  }; |

장점

- 성능상의 이점을 지닌다.

- lock(), unlock()함수는 가상함수 이지만, 컴파일러는 이 메소드를 정적으로 확인할 수 있다.

- 이렇게 선언한 스레드에 안전한 문자열 클래스들은 컴파일 시에 특정 동기화 클래스가 됨, 동적 바인딩을 수행하지 않음

- 컴파일러는 이 함수 호출들을 인라인으로 만들어 낼 수 있다.

단점

- 동기화 메커니즘에 따라 별도의 문자열 클래스 코드를 생성해야 하므로 재사용성이 떨어진다.

**상속**

어떤 동기화 메커니즘을 사용할 것인가를 생성자 인자에서 받아들이도록 한다.

생성자에서 받아들인 Locker 포인터에 따라 어떠한 동기화 구조라도 사용할 수 있다.

|  |
| --- |
| class ThreadSafeString : public std::string {  public:  ThreadSafeString(const char\* s, Locker \*locker) : std::string(s), locker\_(locker) {}  int length()  {  locker\_->lock();  int len = std::string::length();  locker\_->unlock();  return len;  }  private:  Locker\* locker\_;  };  CriticalSectionLock cs;  ThreadSafeString csString("hello", &cs); |

장점

- 하드 코딩과 비교하면 코드가 간단하다.

단점

- lock(), unlock() 가상 호출을 오직 실행 시에만 확인할 수 있고 이로 인해 인라인을 만들 수 없으므로, 인과 성능 부하를 낳을 수 있다.

**템플릿**

템플릿 기반 디자인은 재사용과 효율성을 최대한 누릴 수 잇는 방법이다. ThreadSafeString은 Locker 템플릿 인자를 매개 변수로 받아들이는 템플릿으로 구현된다.

|  |
| --- |
| template<class LOCKER>  class ThreadSafeString : public std::string {  public:  ThreadSafeString(const char\* s) : std::string(s) {}  int length()  {  locker\_.lock();  int len = std::string::length();  locker\_.unlock();  return len;  }  private:  LOCKER locker\_;  };  ThreadSafeString<CriticalSectionLock> csString("hello"); |

- ThreadSafeString을 선언할 때 템플릿 인스턴스가 생성되면서 특정 형태의 동기화가 선택된다.

- 하드 코딩과 같이 이 경우에는 컴파일러가 가상 호출을 확인할 수 있고, 인라인으로 만들 수 있다.

장점

- 인라인을 가능하게 만든다.

**키포인트**

- 가상 함수의 부하는 실행 시에 동적으로 바인딩 되는 함수 호출들이 인라인이 될 수 없기 때문에 발생한다. 잠재적으로 효율성을 증가시킬 수 있는 유일한 방법은 인라인을 통하여 속도를 개선 시킬 수 있다는 것이다. 호출과 반환 오버헤드가 중요하지 않은 함수에 대해서는 인라인으로 인해 얻을 수 있는 효율성이 그다지 높지 않다.

- 템플릿은 상속 계층보다 더욱 성능 친화적이다. 템플릿을 사용하면 컴파일 시에 형식 확인이 이루어지며, 컴파일 시에 발생하는 부하는 없다.