**항목 11. operator=에서는 자기대입에 대한 처리가 빠지지 않도록 하자**

이슈 1.

**자기대입 (self-assignment)이란, 객체가 자기 자신에 대해 대입 연산자를 사용하는 것**

|  |
| --- |
| class Widget {};  Widget w;  w = w; |

문제가 코드이나 컴파일에는 문제가 없다. 자기 대입에 가능성이 있는 코드도 존재

|  |
| --- |
| a[i] = a[j]; // 자기대입의 가능성이 존재  \*ax = \*ay; // 자기대입의 가능성이 존재, 여러 곳에서 하나의 객체를 참조하는 중복참조 시 |

**해결 1.**

같은 타입으로 만들어진 객체 여러 개를 참조자 혹은 포인터로 물어 놓고 동작하는 코드를 작성할 때는 같은 객체가 사용될 **가능성을 고려하는 것이 일반적인 해결법**이다.

같은 기본클래스를 기반으로 만들어진 객체라 해도 **같은 타입으로 선언할 필요까지는 없다**. 파생 클래스 타입의 객체를 참조하거나 가리키는 용도로 기본 클래스의 참조자나 포인터를 사용하면 된다.

|  |
| --- |
| class Base {};  class Derived : public Base {};  void doSomething(const Base& rb, Derived\* pd); // rb 및 \*pd는 원래 같은 객체 였을 수 있음 |

이슈 2.

사용자는 자원 관리 용도로 항상 객체를 만들어야 하며, 만들 어진 자원 관리 객체들이 복사되도록 구현 되어져 있다면 이 때 조심해야 되는 것이 대입 연산자 이다.

자기 대입에 대해 잘 동작하고 있지만, 자원 관리에 의해 사용하기 전에 자원이 해제되어 버릴 수도 있다.

동적 할당된 비트맵을 가리키는 포인터를 멤버로 갖는 클래스가 있다.

|  |
| --- |
| class Bitmap {};  class Widget {  private:  Bitmap \*pb; // 힙에 할당한 객체를 가리키는 포인터  }; |

의미 적으로는 문제가 없을 것 같지만 자기 참조의 가능성이 있는 operator=를 구현해 보겠다.

|  |
| --- |
| class Bitmap {};  class Widget {  Widget& operator=(const Widget& rhs) // 안전 하지 않게 구현된 operator=  {  delete pb; // 현재의 비트맵 사용을 중지  pb = new Bitmap(\*rhs.pb); // 이제 rhs의 비트맵을 사용하도록 만듬  return \*this;  }  private:  Bitmap \*pb; // 힙에 할당한 객체를 가리키는 포인터  }; |

위 코드의 자기 참조 문제는 operator= 내부에서 \*this(대입되는 대상)와 곤가 같은 객체일 가능성이 있다는 것이다. 이 둘이 같은 객체 라면 delete 연산자가 \*this 객체의 비트맵에만 적용되는 것이 아니라 **rhs객체까지 적용되어 버릴 수 있다.**

그러므로, 이 함수가 끝나는 시점에는 해당 객체는 자신의 포인터 멤버를 통해 물고 있던 객체가 삭제되어 버릴 수 있다는 것이다.

**해결 1.**

전통적인 방법으로는 operator=의 첫 머리에서 **일치성 검사(identity test)를 통해 자기대입을 점검**

|  |
| --- |
| class Bitmap {};  class Widget {  Widget& operator=(const Widget& rhs)  {  if (this == &rhs) return \*this; // 객체가 같다면 아무 동작도 수행하지 않음  delete pb;  pb = new Bitmap(\*rhs.pb);  return \*this;  }  private:  Bitmap \*pb; // 힙에 할당한 객체를 가리키는 포인터  }; |

하지만 이 방법도 문젯거리를 안고 있습니다. **new Bitmap 부분인데, 이 부분에서 예외가 터지게 되면 (동적 할당에 필요한 메모리 부족이나, Bitmap 클래스 복사 생성자에서 예외를 던지는 경우) Widget 객체는 결국 처음 실행된 delete pb 수행되어 삭제된 Bitmap 포인터만 남게 됩니다.**

**해결 2.**

“많은 경우에 문장 순서를 세심하게 바꾸는 것만으로 예외에 안전한 코드가 만들어진다” 라는 법칙을 적용할 수 있다.

**pb를 먼저 삭제하지 말고 이 포인터가 가리키는 객체를 복사한 직후에 삭제하면 해결 된다**.

|  |
| --- |
| class Bitmap {};  class Widget {  Widget& operator=(const Widget& rhs)  {  Bitmap \*pOrig = pb; // 원래의 pb를 어딘가에 기억  pb = new Bitmap(\*rhs.pb); // 다음 pb가 \*pb의 사본을 가리키게 만듬  delete pOrig; // 원래의 pb를 삭제  return \*this;  }  private:  Bitmap \*pb;  }; |

‘new Bitmap 부분에서 예외가 발생하더라도, pb는 변경되지 않은 상태가 유지된다. 일치성 검사가 없음에도 이 코드는 자기대입현상을 처리할 수 있다.

**해결 3.**

**복사 후 맞바꾸기(copy and swap) 기법**

|  |
| --- |
| class Bitmap {};  class Widget {  void swap(Widget& rhs) {  } // \*this의 데이터 및 rhs의 데이터를 교환  Widget& operator=(const Widget& rhs)  {  Widget temp(rhs); // rhs의 데이터에 대해 사본을 만듬  swap(temp); // \*this의 데이터를 그 사본의 것과 교환  return \*this;  }  private:  Bitmap \*pb;  }; |

이 방법은 C++의 두 가지 특징으로 조금 다르게 구현될 수 있다.

1. **클래스의 복사 대입 연산자는 인자를 값으로 취하도록 선언하는 것이 가능**

2. **값에 의한 전달을 수행하면 전달된 대상의 사본이 생김**

|  |
| --- |
| Widget& operator=(Widget rhs) // 갑에 의한 전달  {  swap(rhs); // \*this의 데이터를 그 사본의 것과 교환  return \*this;  } |

객체를 복사하는 코드가 함수 본문으로부터 매개변수의 생성자로 옮겨졌기 때문에, **컴파일러가 더 효율적인 코드를 생성**할 수 있다.

**요약.**

\* operator=을 구현할 때, 어떤 객체가 그 자신에 대입되는 경우를 제대로 처리하도록 만듭시다.

원본 객체와 복사 대상 객체의 주소를 비교 해도 되고, 문자의 순서를 적절히 조정할 수도 있으며, 복사 후 맞바꾸기 기법을 써도 된다.

\* 두 개 이상의 객체에 대해 동작하는 함수가 있다면, 이 함수에 넘겨지는 객체들이 사실 같은 객체인 경우에 정확하게 동작하는지 확인해 보세요