Chapter 5. 임시 객체(Temporary)

- 임시 객체의 생성은 성능에 적은 영향을 주는 개념에 속하지 않는다.

- 임시 객체의 기원, 부하, 제거 방법을 알지 못하면 효율적인 코드를 작성하기 힘들다.

- 컴파일러는 조용히 임시 객체를 만들며, 소스 코드에는 존재하지 않는다.

- 컴파일러가 생성한 코드에서 임시 객체가 생길 만한 가능성을 지닌 몇 가지 예제를 다룬다.

**객체 정의**

- 클래스 Ratinal이 다음과 같이 선언

- Rational 형식의 객체의 인스턴스를 몇 가지 동일한 방법으로 만들 수 있다

|  |
| --- |
| class Rational  {  friend Rational operator+(const Rational& a, const Rational& b);  public:  Rational(int a = 0, int b = 1) : m(a), n(b) {}  private:  int m;  int n;  };  Rational r1(100);  Rational r2 = Rational(100);  Rational r3 = 100; |

- **r1 형식 만이 임시 객체를 생성하지 않는다고 보장**할 수 있다.

- r2, r3는 컴파일러에 따라 임시 객체를 생성할지 모른다.

- r3의 경우, 컴파일러는 Rational(int, int) 생성자를 사용해 정수 100을 Rational 형식의 임시 객체를 생성한다.

- 다음 복사 생성자를 사용해 새롭게 생성된 임시 객체로부터 r3를 초기화 후 임시 객체를 제거한다.

- 부하는 **임시 객체의 생성과 소멸**이다.

|  |
| --- |
| {  Rational \_temp;  \_temp.Rational::Rational(100, 1); // 임시 객체 생성  r3.Rational::Rational(\_temp); // r3에 \_temp 객체를 복사 생성  \_temp.Rational::~Rational(); // 임시 객체 제거  } |

참고, 대부분의 컴파일러 최적화를 통해 임시 객체를 없애기 때문에, 셋 모두 같은 효율성을 지닐 수 있다.

**형식 불일치**

- 형식 불일치의 일반적인 경우에는 형식 x의 객체가 필요할 때마다 어떤 다른 형식이 제공 되는 것.

- 컴파일러는 제공된 형식을 형식 x의 객체로 변환한다. 임시 객체는 이 과정에서 생겨난다.

|  |
| --- |
| {  Rational r;  r = 100;  } |

- Rational 클래스는 정수 매개 변수를 받아들일 연산자를 선언하지 않았으므로, Rational 객체를 기대하게 된다.

- 컴파일러는 제공 받은 정수 인자를 Rational 형식의 객체로 변환

- Rational 클래스의 경우 이 정수를 처리할 생성자가 존재한다. (없다면, 컴파일과정에서 에러가 발생하므로 컴파일 시점에서 캐치가 가능할 것)

|  |
| --- |
| Rational(int a = 0, int b = 1) : m(a), n(b) {}  {  Rational \_temp;  \_temp.Rational::Rational(100, 1); // 임시 객체 생성  r.Rational::Rational(\_temp); // r에 \_temp 객체를 복사 생성  \_temp.Rational::~Rational(); // 임시 객체 제거  } |

- 이러한 형식 변환을 수행하지 못하게 하려면 생성자를 explicit 키워드를 선언하여 사용하면 된다.

- explicit 키워드를 사용하면 컴파일러는 이 생성자를 변환 생성자로 사용하지 못한다.

|  |
| --- |
| explicit Rational(int a = 0, int b = 1) : m(a), n(b) {} |

- 대신 정수를 인자로 받아 들이도록 **대입 연산자를 선언하여 임시 객체를** 없앨 수 있다.

|  |
| --- |
| Rational& operator=(int a) { m = a; n = 1; return \*this; }; |

**값으로 전달**

- 객체를 값으로 전달 할 때, 다음과 같이 형식 매개 변수를 실제 매개 변수로 초기화한다.

|  |
| --- |
| **T formalArg = actualArg;** |

- T는 클래스 형식, g()가 T 인자를 받아들이는 함수라고 가정, g()는 아래 처럼 호출될 것

|  |
| --- |
| **void g ( T formalArg ) {}**  **T t;**  **g(t);** |

- g()는 지역 인자 formalArg를 위해 개체 틀을 스택에 가지고 있다.

- 컴파일러는 t의 내용을 g()가 스택에 가지고 있는 formalArg에 복사해야한다.

- 이것을 수행하는 **방법은 임시객체를 생성하는 것이다**.

- 컴파일러는 형식 T의 임시 객체를 생성한 다음, t를 입력 인자로 사용해, 임시 객체를 복사 생성한다.

- 이 새롭게 생성된 임시 객체는 g()에 참조로 전달된다.

|  |
| --- |
| T \_temp;  \_temp.T::T(t); //t\_로 부터 \_temp를 복사 생성  g(\_temp); // \_temp를 참조로 전달  \_temp.T::~T(); // 임시 객체 소멸 |

- 임시 객체 생성 소멸은 부담이 간다.

- **가능 하다면 객체들을 포인터나 참조로 전달하여 임시 객체를 방지하자**

**값으로 반환**

- 값으로 객체를 반환하게 되면 임시 객체를 생성하게 될 것이다.

- f()의 반환 값은 string 객체이며, 임시 객체는 반환 값을 저장하기 위해 생성된다.

|  |
| --- |
| std::string f()  {  std::string s;  // s의 연산  return s;  }  string p = f(); |

또, 다음 같은 경우 임시 객체를 생성하지 않는다.

|  |
| --- |
| string s1 = “hello”;  string s2 = “world”;  1. string s3 = s1 + s2; // 임시 객체 생성 x  2. string s3;  s3 = s1 + s2 // 임시 객체 생성 |

**op=()를사용하여 임시 객체 없애기**

- s3를 초기화 하기 위해 다음과 같은 동작을 수행한다고 했을 때, operator=(), operator+=() 를 사용하여 임시 객체를 없앨 수 있다.

|  |
| --- |
| s3 = s1 + s2; // 임시 객체 생성  수정 후,  s3 = s1; // operator=()  s3 += s2; // operator+=() |

**키 포인트**

- 임시 객체는 생성자와 소멸자 실행 때문에 인과 성능 부하를 두 배로 만들어 낸다.

- 생성자에 explicit을 선언하면 컴파일러는 이 생성자를 사용하여 이면에서 형식 변환을 수행하지 못한다.

- 컴파일러는 형식 불일치를 해결하기 위해 임시 객체를 만든다. 함수 오버로딩을 통하여 임시 객체 생성을 방지할 수 있다.

- 가능하다면 객체 복사를 방지하자. 참조로 전달하고 반환한다.

- <op>= 연산자를 사용하여 임시 객체를 제거할 수 있다. 여기에서 <op>는 +. -, \*, /이 될 수 있다.