Charter 4. 반환 값 최적화 (Return Value Optimization)

컴파일러는 소스 코드를 변환하여 객체 생성을 없애는 방식을 사용한다. 이 최적화를 반환 값 최적화(RVO)라고 한다.

**값으로 반환의 역학**

Complex 클래스는 복소수의 표현

|  |
| --- |
| class Complex {  friend Complex operator+(const Complex& a, const Complex& b)  {  Complex retVal;  retVal.real = a.real + b.real;  retVal.imag = a.imag + b.imag;  return retVal;  }  public:  // 기본생성자  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}  // 복사 생성자  Complex(const Complex& c) : real(c.real), imag(c.imag) {}  // 대입연산자  Complex& operator= (const Complex& c);  private:  double real;  double imag;  }; |

C1, c2, c3이 Complex이고,

C3 = C1 + C2;

- C1 + C2의 값은 어떻게 C3로 대입할 수 있을까. 컴파일러가 사용하는 한 가지 유명한 방법은 임시 객체를 생성하여 Complex::operator+()의 세 번째 인자로 전달하는 것이다. 이 임시 객체(\_\_result)는 참조로 전달된다.

그러므로 컴파일러는 다음 코드를

|  |
| --- |
| Complex& Complex::operator+(const Complex& c1, const Complex& c2) |

약간 다른 함수로 고쳐 쓴다.

|  |
| --- |
| void complex\_Add(const Complex& \_\_result, const Complex& c1, const Complex& c2) |

즉, c3 = c1 + c2 는 다음과 같이 변환된다.

Struct Complex \_\_tempResult;

Complex\_Add(\_\_tempResult, c1, c2);

c3 = \_\_tempResult

- 값으로 반환 구현은 지역 객체 retVal을 없애고, 반환 값을 \_\_tempResult 임시 객체로 직접 계산하는 형태로 최적화 할 수 있다. 이것이 반환 값 최적화 이다.

**반환 값 최적화**

어떠한 최적화도 없다면 Complex\_Add()에 대해 컴파일러가 생성한 코드는 다음과 같다.

|  |
| --- |
| void Complex\_Add(const Complex& \_\_tempResult, const Complex& c1, const Complex& c2)  {  struct Complex retVal;  retVal.Complex::Complex(); // retVal을 생성한다.  retVal.real = c1.real + c2.real;  retVal.imag = c1.imag + c2.imag;  \_\_tempResult.Complex::Complex(retVal);  retVal.Complex::Complex(); // retVal 객체 소멸  return;  } |

컴파일러는 지역 객체 retVal을 없애고, 이것을 \_\_tempResult로 교체하여 Complex\_Add()를 최적화할 수 있다. 이것이 반환 값 최적화 이다.

|  |
| --- |
| void Complex\_Add(const Complex& \_\_tempResult, const Complex& c1, const Complex& c2)  {  \_\_tempResult.Complex::Complex();  \_\_tempResult.real = c1.real + b.real;  \_\_tempResult.imag = c1.imag + b.imag;  return;  } |

RVO는 지역 retVal 객체를 없앰으로써 소멸 연산뿐만 아니라, 생성 연산도 절약하였다.

- 컴파일러 최적화는 자연적으로 원본 연산의 정확성을 유지해야 한다. RVO의 경우에는 원본 연산의 정확성을 항상 유지하기가 쉽지 않다. RVO가 필수는 아니기 때문에, 컴파일러는 복잡한 함수에 대해서 RVO를 적용하지 않을 것이다.

ex) 만약 함수가 여러 return 구문에서 다른 이름을 가진 객체들을 반환한다면, RVO는 적용되지 않을 것이다 RVO를 적용할 확률을 높이려면 동일한 이름을 가진 객체를 반환해야 한다.

버전1. RVO 적용되지 않음

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  Complex retVal;  retVal.real = a.real + b.real;  retVal.imag = a.imag + b.imag;  return retVal;  } |

버전2. RVO 적용

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  double r = a.real + b.real;  double i = a.imag + b.imag;  return Complex(r, i);  } |

버전3. RVO 적용되지 않음

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  Complex retVal(a.real + b.real, a.imag, b.imag);  return retVal;  } |

버전4. RVO 적용

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  return Complex(a.real + b.real, a.imag, b.imag);  } |

버전1은 명명된 변수(retVal)를 반환 값으로 사용한 반면, 버전2는 이름이 지정되지 않은 변수를 사용하엿다는 차이점,

이것 때문에 컴파일러의 선택이 달라졌을지도 모른다. 버전2는 반환 구문에서 생성자 호출을 사용하였지만 이름을 지정하지는 않았다. 이 특정 컴파일러 구현은 명명된 변수를 최적화하지 않는 경우에 속한다.

RVO는 버전4.에는 적용되었지만 버전 3에는 적용되지 않았다.

더불어 반환 값 최적화를 적용하려면 복사 생성자도 정의해야 한다. 만약 연관된 클래스가 복사 생성자를 정의하지 않는다면, RVO는 비활성화 된다.

**연산 생성자**

컴파일러가 RVO를 적용시키지 못한다면, 연산 생성자를 사용할 수 있다.. 컴파일러는 버전 1에 RVO를 적용하지 않았다.

버전1.

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  double r = a.real + b.real;  double i = a.imag + b.imag;  return Complex(r, i);  } |

- 이 구현은 기본 Complex 객체를 생성하고 이것의 멤버 필드 설정을 이후에 하고 있다. 객체를 생성한 다음, 입력 객체로부터 제공받은 정보로 멤버 데이터의 값을 채워 넣는다. Complex retVal 객체를 생성한 다음 여러 단계를 거친 후에 반환한다. 연산 생성자는 이 단계를 하나로 합쳐서 명명된 지역 변수를 없앤다.

버전5.

|  |
| --- |
| friend Complex operator+(const Complex& a, const Complex& b)  {  return Complex(a, b);  } |

- 버전 5에서 사용된 연산 생성자는 두 개의 입력 인자를 더하여 새로운 Complex 객체를 생성한다.

|  |
| --- |
| Complex(const Complex& x, const Complex& y) : real(x.real + y.real), imag(x.imag + y.imag) {} |

- 컴파일러는 버전1 보다 버전5에 RVO를 적용할 확률이 높아졌다.

**키포인트**

- 객체를 값으로 반환해야 하는 경우, 반환 값 최적화를 사용하면 지역 객체가 생성하고, 소멸할 필요가 없어지므로 성능이 좋아질 것이다.

- 컴파일러의 구현에 따라 RVO의 적용 사례가 달라진다. 언제 그리고 어떤 조건에서 RVO가 적용되는지 알려면 컴파일러 문서를 참조하거나 직접 실험해 보아야 한다.

- 연산 생성자를 사용하면 RVO가 적용될 확률을 높일 수 있다.