Chapter 12. 참조 횟수

- 시간이 지날수록 디자인과 구현의 명확함은 유지, 보수와 빈번한 릴리즈 주기 때문에 서서히 사라진다. 소프트웨어가 혼란스러워 지는 것은 불가피한 일이다.

- 혼란한 소프트웨어가 가진 주요한 난점 중 하나는 메모리 오용이다.

- 할당된 메모리는 포인터 전달을 통해 시스템 사이에서 표류한다. 포인터는 모듈과 스레드 사이에서 전달된다.

**혼란한 소프트웨어의 주요 난점**

- 메모리 누수 : 메모리가 해지 되지 않았을 때 발생한다. 메모리 누수는 시간이 지날수록 메모리의 점유가 정도를 넘어서게 되었을 때 응용프로그램을 다운시킬 수 있다.

- 너무 빠른 삭제 : 메모리 포인터의 소유가 명확하지 않을 때, 이 메모리가 삭제된 후에 액세스 되는 경우가 발생한다. 이렇게 되면 엄청난 오류가 발생한다.

- C++에서는 두 가지 문제의 해결책을 제시한다. C++은 객체 생성, 소멸, 복사, 대입의 모든 지점을 제어할 수 있다.

- 참조 횟수라고 불리는 가비지 컬렉션을 개발하여 이러한 제어를 조정할 수 있다. 객체는 자신에 대한 현재 참조 횟수를 기록하고 있고, 참조 횟수가 0이되면 자신을 소멸한다. 다른 말로, 아무도 자기 자신을 더 이상 사용하지 않게 되면 객체는 소멸된다.

- 소프트웨어 결함의 상당수가 메모리 오용임을 볼 때, 참조 횟수는 C++의 매우 중요한 기술 이다.

- 참조 횟수는 성능 최적화의 일종, CPU 주기 이외에 메모리 점유도 줄여 줄 수 있다.

|  |
| --- |
| class MyString  {  public:  MyString() {}  MyString & operator=(const char\* rhs)  {  pData = const\_cast<char\*>(rhs);  }  MyString & operator=(const MyString& rhs)  {  if (this == &rhs)  return \*this;  delete[] pData;  int length = strlen(rhs.pData) + 1; // 종료 널을 포함한다.  pData = new char[length]; // 새로운 배열에 대한 공간을 만든다.  memcpy(pData, rhs.pData, length); // 우변의 객체로 부터 문자들을 복사한다.  return \*this;  }  private:  char\* pData;  };  int main()  {  MyString p, s;  p = s = "TWO";  return 0;  } |

- 위 예쩨는 힙 메모리를 할당하고 문자열을 복사하는데 드는 CPU주기 뿐만 아니라, 메모리 사용의 관점에서도 명백한 낭비이다. 이론적으로 두 객체가 동일한 메모리 리소스를 가리킬 수 있도록 참조 횟수를 사용한다면 더욱 부하가 적을 것

- **참조 횟수가 메모리 사용을 줄여준다는 것은 의심이 필요 없는 사실**

- 이장의 주제는 **이것이 실행 속도도 증가시켜 줄 수 있는 가** 이다.

**구현 세부 사항**

- More Effective C++의 Item 29의 참조 횟수 구현을 기반으로 성능 비교

|  |
| --- |
| class Widget  {  public:  Widget(int size){}  Widget(const Widget& rhs){}  ~Widget(){}  Widget& operator = (const Widget& rhs) {}  void doThis() {};  int showThat() const { return refCount; }  private:  char\* somePtr;  int refCount; // 참조 카운트 멤버 변수  };  class RCWidget // 참조 횟수를 가진 Widget 클래스  {  public:  RCWidget(int size) :value(new Widget(size)) {}  void doThis() { value->doThis(); }  int showThat() const { return value->showThat(); }  private:  Widget \* value;  }; |

- refCount가 참조 횟수를 기록하는 멤버

- refCount가 0이 되면 Widget 객체는 자신을 소멸

- RCWidget 클래스는 Widget 프록시의 역할을 수행한다. 실제 작업을 자신이 가리키는 Widget 객체로 전달한다.

- BigInt클래스를 인용한 예제, BigInt 클래스는 양의 정수를 이진 코드 십진수로 표현

|  |
| --- |
| class BigInt  {  friend BigInt operator+ (const BigInt& lhs, const BigInt& rhs)  {  return BigInt(lhs, rhs);  }  public:  BigInt(const char\* s)  {  if (s[0] == '\0')  s = "0";  size = ndigits = strlen(s);  digits = new char[size];  for (unsigned i = 0; i < ndigits; ++i)  digits[i] = s[ndigits - 1 - i] - '0';  }  BigInt(unsigned u = 0)  {  unsigned v = u;  for (ndigits = 1; (v /= 10) > 0; ++ndigits) {}  size = ndigits;  digits = new char[size];  for (unsigned i = 0; i < ndigits; ++i)  {  digits[i] = u % 10;  u /= 10;  }  }  BigInt(const BigInt& copyFrom) // 복사 생성자  {  size = ndigits = copyFrom.ndigits;  digits = new char[size];  for (unsigned i = 0; i < ndigits; ++i)  digits[i] = copyFrom.digits[i];  }  BigInt& operator= (const BigInt& rhs)  {  if (this == &rhs)  return \*this;  ndigits = rhs.ndigits;  if (ndigits > size)  {  delete[] digits;  size = ndigits;  digits = new char[size];  for (unsigned i = 0; i < ndigits; ++i)  digits[i] = rhs.digits[i];  return \*this;  }  }  BigInt& operator+= (const BigInt& rhs)  {  unsigned max = 1 + (rhs.ndigits > ndigits ? rhs.ndigits : ndigits);  if (size < max)  {  size = max;  char\* d = new char[size];  for (unsigned i = 0; i < ndigits; ++i)  d[i] = digits[i];  delete[] digits;  digits = d;  }  while (ndigits < max)  digits[ndigits++] = 0;  for (unsigned i = 0; i < ndigits; ++i)  {  digits[i] += rhs.fetch(i);  if (digits[i] >= 10)  {  digits[i] -= 10;  digits[i + 1] += 1;  }  }  if (digits[ndigits - 1] == 0)  --ndigits;  return \*this;  }  ~BigInt()  {  delete[] digits;  }  private:  char\* digits;  unsigned ndigits;  unsigned size; // 할당된 문자열의 크기  BigInt(const BigInt& left, const BigInt& right) // 연산 생성자  {  size = 1 + (left.ndigits > right.ndigits ? left.ndigits : right.ndigits);  digits = new char[size];  ndigits = left.ndigits;  for (unsigned i = 0; i < ndigits; ++i)  digits[i] = left.digits[i];  \*this += right;  }  inline char fetch(unsigned i) const  {  return i < ndigits ? digits[i] : 0;  }  }; |

- BigInt는 자신의 자릿수를 저장하기 위해 힙 메모리를 사용하기 때문에 참조 횟수에 적용하기 좋다.

- BigInt의 생성자, 소멸자, 대입 연산자에 대해 구성 되어 있다.

- 참조 횟수를 사용하기 위해 모든 BigInt 객체와 참조 횟수를 연관시켜야 한다.

- refCount 멤버를 BigInt에 바로 추가 해도 되고, 기본 클래스로부터 파생 시켜도 된다.

- RCObject 클래스는 참조 횟수를 가진 객체의 기본클래스이며, 참조 횟수 변수와 조작을 캡슐화 하고 있다.

|  |
| --- |
| {  public:  void addReference() { ++refCount; }  void removeRefCount() { if (--refCount == 0) delete this; }  void markunshareable() { shareable = false; }  bool isshareable() const { return shareable; }  bool isshared() const { return refCount > 1; }  protected:  RCObject() : refCount(0), shareable(true) {}  RCObject(const RCObject& rhs) : refCount(0), shareable(true) {}  RCObject& operator= (const RCObject& rhs) { return \*this; }  virtual ~RCObject() {}  private:  int refCount;  bool shareable;  }; |

- RCObject 클래스로부터 상속하도록 BigInt 클래스를 수정한다.

|  |
| --- |
| class BigInt : public RCObject |

- 이제 Bigint의 참조 횟수를 가진 구현인 RCBigInt로 발전하였다.

- 실제 BigInt를 가리킬 필요가 있는데, 실제 포인터나 스마트 포인터를 사용하면 된다.

- 이 특정 스마트 포인터는 참조 횟수를 기록하는 역할을 수행한다.

|  |
| --- |
| template<class T>  class RCPtr  {  public:  RCPtr(T\* realPtr = nullptr) : pointee(realPtr) { init(); }  RCPtr(const RCPtr& rhs) : pointee(rhs.pointee) { init(); }  ~RCPtr()  {  if (pointee)  pointee->removeReference();  }  RCPtr& operator=(const RCPtr& rhs)  {  if (pointee != rhs.pointee)  {  if (pointee)  pointee->removeReference();  pointee = rhs.pointee;  init();  }  return \*this;  }  T\* operator->() const { return pointee; }  T& operator\*() const { return \*pointee; }  private:  T \* pointee;  void init()  {  if (pointee == nullptr)  return;  if (pointee->isshareable())  pointee = new T(\*pointee);  pointee->addReference();  }  }; |

- 마지막으로, 참조 횟수를 가진 BigInt를 만들기 위해서 나머지를 구현한다.

- 참조 횟수를 가진 BigInt의 이름을 RCBigInt라고 부른다. RCBigInt는 매우 직관적이다.

|  |
| --- |
| class RCBigInt  {  friend RCBigInt operator+ (const RCBigInt& left, const RCBigInt& right)  {  return RCBigInt(\*(left.value) + \*(right.value));  }  public:  RCBigInt(const char\* p) : value(new BigInt(p)) {}  RCBigInt(unsigned u = 0) : value(new BigInt(u)) {}  RCBigInt(const BigInt& bi) : value(new BigInt(bi)) {}  private:  RCPtr<BigInt> value;  }; |

- 성능 테스트에서 RCBigInt 객체를 비번하게 대입하고 복사하는 경우 RCBigInt가 우수하다는 것에 틀림없을 것이다.

- 반면에 새로운 RCBigint 객체에 대한 첫 번째 참조를 생성하는 새로운 RCBigInt는 평범한 Bigint와 비교 했을 때 좀더 많은 부하를 준다.

- 즉, 참조 횟수가 성능 이득과 유의어가 아니라는 증거가 된다. 이것은 참조 횟수가 성능에 부정적인 영향을 끼친 경우 이며, 반대로 성능을 증가 시킨 경우도 있다.

**참조 횟수와 실행 속도와의 관계는 상황에 따라 다르며, 아래 몇가지 요소의 영향을 받는다.**

- 대상 객체가 얼마나 많은 리소스를 점유하는가? 만약 대형 객체가 많은 양의 메모리를 차지하는 경우, 메모리를 보호하는데 실패하면 가용 메모리에 한계가 생겨나고 캐시 누락과 페이지 실패라는 극단적인 성능 저하가 발생

- 대상 객체가 사용하는 리소스를 할당하는데 얼마나 많은 부하가 생기는가?

- 얼마나 많은 객체가 대상 객체의 단일 인스턴스를 공유할 가능성이 있는가? 대입 연산자와 복사 생성자를 사용함에 따라 공유가 증가한다.  
- 얼마나 자주 대상 객체에 대한 첫 번째 참조를 생성 시키는가?

- 극단적으로 작은 수의 대상 객체들을 공유하며 참조 횟수가 많은 인스턴스들을 가지게 될 수도 있다. 하지만 첫 번째 참조와 마지막 참조를 생성하고 소멸하는 것은 제한되어 있다. 대상 객체는 제한된 리소스를 많이 점유하고 있으며, 제한 리소스를 획득하고 해지하는 것은 부하를 많이 준다. 이런 경우는 **참조 횟수가 성능을 증가 시켜주는 경우가** 된다.

- 반대로, 대상 객체가 많이 생성되나 참조 횟수가 1, 2 이상 되지 않는 경우가 있다. 이 경우의 **참조 횟수는 실행 시간을 늘일 수 있다**.

- 다음에서 참조 횟수를 가진 객체와 평범한 객체 사이에서 성능 조화에 영향을 주는 또 다른 구현 방안을 살펴 본다.

**미리 존재하는 클래스**

- BigInt에 대한 참조 카운터를 추가하기 위해 BigInt를 RCObject로부터 상속

- 이제 BigInt 구현에 손을 댈 수 가 없기 떄문에, 참조 횟수를 저장하고 조작하기 위해 별도의 클래스가 필요

- Meyers는 이 클래스를 CountHolder라고 하였다.

- BigInt 클래스로부터 참조 횟수를 분리 하는 것이외에 RCPtr 객체 대신에 RCIPtr 포인터를 사용. RCIPtr은 RCPtr과 같은 스마트 포인터 이지만, CountHolder를 가리키는 포인터를 통하여 BigInt를 간접적으로 가리킨다.(RCPtr은 BigInt를 직접 가리킴)

구현 관점에서의 세 가지 고려 항목

1. BigInt 클래스는 원래 형태로 돌아가고 더 이상 RCObject에서 상속 받지 않는다.

2. RCBigInt 클래스는 이제 RCPtr 대신 RCIPtr 멤버를 가지고 있다.

3. RCIPtr를 구현해야 한다. RCIPtr은 CountHolder를 중첩 클래스로 만든다.

|  |
| --- |
| class RCBigInt // 이전과 같다.  {  friend RCBigInt operator+ (const RCBigInt& left, const RCBigInt& right)  {  return RCBigInt(\*(left.value) + \*(right.value));  }  public:  RCBigInt(const char\* p) : value(new BigInt(p)) {}  RCBigInt(unsigned u = 0) : value(new BigInt(u)) {}  RCBigInt(const BigInt& bi) : value(new BigInt(bi)) {}  private:  RCPItr<BigInt> value; // RCIPtr을 사용한다.  }; |

|  |
| --- |
| template <class T>  class RCIPtr  {  public:  RCIPtr(T\* realPtr = nullptr) : counter(new CountHolder())  {  counter->pointee = realPtr;  init();  }  RCIPtr(const RCIPtr& rhs) : counter(rhs.counter)  {  init();  }  ~RCIPtr()  {  if (counter)  counter->removeReference();  }  RCIPtr& operator=(const RCIPtr& rhs)  {  if (counter != rhs.counter)  {  if (counter)  counter->removeReference();  counter = rhs.counter;  init();  }  return \*this;  }  T\* operator->() const { return counter->pointee; }  T& operator\*() const { return \*(counter->pointee); }  private:  struct CountHolder : public RCObject  {  ~CountHolder() { delete pointee; }  T\* pointee;  };  RCIPtr<T>::CountHolder \*counter;  void init()  {  if (counter == nullptr)  return;  if (counter->isshareable() == false)  {  counter = new CountHolder();  counter->pointee = new T(\*counter->pointee);  }  counter->addReference();  }  }; |

- RCIPtr<T>::operator= 의 구현은 부하 면에서 RCPtr의 해당 구현과 유사

- 대입 테스트의 실행 속도는 RCBigInt의 두 버전이 동일

- 생성 테스트는 첫 번째 BigInt 참조의 생성과 소멸의 부하가 더욱 심해 졌다. 힙 기반의 새 BigInt 객체를 생성해야 할 뿐만 아니라, 새 CountHolder 객체도 생성해야 하며, 소멸 시에 이것들을 삭제 해야 한다. BigInt와 RCBigInt 간의 성능 격차는 이 시나리오에서 더욱 커진다.

**동시 참조 횟수(Concurrent Reference Counting)**

- 참조 횟수의 두 가지 버전을 살펴 보았다.(RCPtr, PCIPtr)

- 세 번째 버전의 참조 횟수를 가진 객체는 **멀티 스레드 환경에서 처리한다.**

- 이 경우 여러 스레드가 참조 횟수를 가진 객체를 동시에 액세스할 수 있다.

- 참조 횟수를 저장하고 있는 변수는 있는 변수는 갱신이 원자적으로 이루어질 수 있도록 보호 되어야 함

- 잠금 클래스 7장.

|  |
| --- |
| class MutexLock  {  public:  MutexLock() { mutex = CreateMutex(NULL, FALSE, NULL); }  virtual ~MutexLock() { CloseHandle(mutex); }  void lock() { WaitForSingleObject(mutex, INFINITE); }  void unlock() { ReleaseMutex(mutex); }  private:  HANDLE mutex;  }; |

- 동시성 제어를 RCBigInt 에 추가 하려면 클래스 선언 중 PCIPtr 선언을 수정해야 한다.

|  |
| --- |
| class RCBigInt  {  friend RCBigInt operator+ (const RCBigInt& left, const RCBigInt& right)  {  return RCBigInt(\*(left.value) + \*(right.value));  }  public:  RCBigInt(const char\* p) : value(new BigInt(p)) {}  RCBigInt(unsigned u = 0) : value(new BigInt(u)) {}  RCBigInt(const BigInt& bi) : value(new BigInt(bi)) {}  private:  RCIPtr<BigInt, MutexLock> value;  }; |

- 동시 액세스 처리하는 작업이 RCIPtr 클래스에 주어졌다. 추가적인 잠금 클래스 인자를 이 템플릿 선언에 확장

|  |
| --- |
| template <class T, class LOCK>  class RCIPtr  {  public:  …  private:  struct CountHolder : public RCObject  {  ~CountHolder() { delete pointee; }  T\* pointee;  LOCK lock; // 잠금 클래스  };  RCIPtr<T, LOCK>::CountHolder \*counter;  …  }; |

- init() 메소드는 동시성 때문에 간섭 받지도 않고, 변하지도 않는다. 필요하다면 init()의 호출자가 원자 액세스를 처리할 수 있다. (기존과 동일)

- 나머지 메소드에서 잠금을 명시적으로 조작한다. 구현은 직관적이다. 직렬화를 필요로 하는 모든 연산은 lock(), unlock()호출을 쌍으로 둘러 싸여진다.

|  |
| --- |
| template <class T, class LOCK>  class RCIPtr  {  public:  RCIPtr(T\* realPtr = nullptr) : counter(new CountHolder())  {  counter->pointee = realPtr;  init();  }  RCIPtr(const RCIPtr& rhs) : counter(rhs.counter)  {  if (rhs.counter) rhs.counter->lock.lock();  init();  if (rhs.counter) rhs.counter->lock.unlock();  }  ~RCIPtr()  {  if (counter)  {  counter->lock.lock();  counter->removeReference();  counter->lock.unlock();  }    }  RCIPtr& operator=(const RCIPtr& rhs)  {  if (counter != rhs.counter)  {  if (counter)  {  counter->lock.lock();  counter->removeReference();  counter->lock.unlock();  }  counter = rhs.counter;  if (rhs.counter) rhs.counter->lock.lock();  init();  if (rhs.counter) rhs.counter->lock.unlock();  }  return \*this;  }  T\* operator->() const { return counter->pointee; }  T& operator\*() const { return \*(counter->pointee); }  private:  struct CountHolder : public RCObject  {  ~CountHolder() { delete pointee; }  T\* pointee;  LOCK lock; // 잠금 클래스  };  RCIPtr<T, LOCK>::CountHolder \*counter;  void init()  {  if (counter == nullptr)  return;  if (counter->isshareable() == false)  {  counter = new CountHolder();  counter->pointee = new T(\*counter->pointee);  }  counter->addReference();  }  }; |

- 원자 갱신을 제공하기 위해 필요한 추가적인 연산은 RCBigInt 대입의 부하를 약간 증가 시키는 것

- 멀티스레드 환경으로 이전한다고 해서 객체 대입에 커다란 영향을 주지는 않는다. 하지만 참조 생성과 소멸에 극적인 효과를 가졌던 것은 사실

- BigInt 참조의 생성(소멸)은 BigInt 객체와 CountHolder 객체를 힙에 생성(소멸)시켜야 하기 때문에 이미 많은 부하를 가지고 있었다.  
- 평범한 BigInt와 참조 횟수를 가진 BigInt와의 성능 격차는 거의 BigInt 객체가 9배 가량 좋은 것으로 판명됨

**키 포인트**

참조 횟수는 자동으로 성능을 증가시켜 주지는 못한다. 성능이 중요한 요소라면 참조 횟수, 실행 속도, 리소스 보존은 주의 깊게 평가되어야 할 섬세한 상호 작용을 이룬다. 참조 횟수는 사용 패턴에 따라 성능을 도와주기도 하고 해치기도 한다. 참조 횟수가 이로운 경우에는 다음 중 어느 한 가지가 두드러진다.

- 대상 객체가 리소스를 많이 차지 한다.

- 필요한 리소스를 할당하고 해지하는데 부하가 많이 든다.

- 공유의 정도가 크다. 복사 생성자와 대입 연산자의 사용 때문에 참조 횟수가 커지기 마련이다.

- 참조의 생성과 소멸이 상대적으로 부담이 적다.

만약, 어떤 상황이 위 항목들과 반대라면, 참조 횟수를 사용하지 않고 참조 횟수를 가지지 않은 평범한 객체를 사용하는 것이 좋다.