**Chapter 1. 전쟁이야기의 추적(Trace)**

- 객체를 정의하면 객체 생성자와 소멸자의 형태로 조용한 실행(silent execution)이 발생한다. 객체 생성과 소멸은 대게 부하가 아니기 대문에 우리는 “조용한 오버 헤드”라는 말 대신 “조용한 실행”이라는 말을 사용한다. 만약 생성자와 소멸자가 수행하는 연산이 항상 필요하다면, 이 연산 들을 효율적인 코드로 간주되어야 한다. (생성자와 소멸자를 인라인으로 만들면 호출과 반환 오버헤드를 줄여줄 수 있다.) 앞에서 보았듯이, 생성자와 소멸자는 항상 순수한 특성만을 가지는 것은 아니고 현저한 오버헤드를 생성할 수 있다. 어떤 상황에서는 생성자, 소멸자가 수행한 연산이 사용되지 않는다. 이것은 C++ 언어적인 주제라기 보다 디자인 주제에 밀접하다. 하지만, C는 생성자/소멸자의 개념을 갖지 않기 때문에 C코드에서 이러한 상황은 잘 발생하지 않는다.

- 객체를 참조로 전달하는 것도 좋은 성능을 보장하지 않는다. 객체 복사를 수행하지 않으면 도움이 되겠지만, 우선 객체를 생성하고 소멸할 필요가 없다면 더욱 도움이 될 것이다.

- 잘사용되지 않는 결과를 만들어 내는 연산을 방지 한다. 추적이 꺼진 경우, string 멤버의 생성은 가치가 없고 부담이 많이 든다.

- 디자인의 유연성에 너무 집착하지 말자. 여러분이 필요로 하는 것은 문제 영역에 적합한 정도로 유연한 디자인이다. char 포인터는 가끔 string에 비해서 더 효율적이면서 단순한 작업을 잘 수행한다.

- 인라인, 작은 함수를 자주 호출하여 발생하는 함수 호출 오버헤드를 제거한다. Trace 생성자와 소멸자를 인라인으로 만들면 Trace 오버헤드를 줄이기 쉬워진다.

**Chapter 2. 생성자와 소멸자**

- 생성자와 소멸자는 손으로 작성한 C코드 만큼 효율적이어야 할 것이다. 하지만, 실제 상황에서 생성자와 소멸자는 과잉연산의 형태를 가진 오버헤드를 종종 포함하고 있다.

- 객체의 생성(소멸)은 부모와 멤버 객체의 재귀적인 생성(소멸)을 발생시킨다. 복잡한 계층에서 객체 생성/소멸 조합을 조심해야 한다. 이것 때문에 생성과 소멸이 더 많은 부하를 낳게 된다.

- 코드가 생성한 모든 객체와 수행한 모든 연산을 실제로 사용하는지 확인한다. 프로그래머는 자신이 사용하는 클래스 내부를 주의 깊게 들여다보기를 권유한다. 이 충고는 OOP가 주장하는 것과는 약간 거리가 있다. OOP는 클래스를 캡슐화된 암흑 상자로 간주하고 내부를 들여다보는 것을 권유하지 않기 때문이다. 위 두 충고간의 상반되는 관점에서 어떻게 균형을 맞춰야 할까? 이 질문에 대한 해답은 상황에 따라 다르기 때문에 간단해질 수는 없다. 암흑 상자 접근이 코드의 80%에 대해서는 완벽하게 작동하겠지만, 성능에 치명적인 20%의 코드에 대해서는 위험요소를 만들 수 있다. 이 문제는 또한 응용프로그램에 의존하기도 한다. 어떤 응용프로그램은 유지-보수성과 유연성에 가중치를 부여하며, 다른 프로그램은 성능을 가장 최우선으로 고려하기도 한다. 프로그래머로서 여러분은 최대화시키고자 하는 사항이 정확하게 무엇인지에 관한 질문에 대답해야 한다

.

- 객체 생명 주기는 전혀 부담을 갖지 않는 것이 아니다. 최소한, 객체의 생성과 소멸은 CPU 주기를 소모한다. 꼭 필요하지 않으면 객체를 생성하지 말자. 전형적으로 객체가 실제로 사용되는 범위까지 객체 생성을 연기 하는 것이 좋다.

- 컴파일러는 생성자 본문으로 들어가기 전에 포함된 객체 멤버를 초기화 한다. 멤버 객체 생성을 완료하기 위해서 초기화 단계를 사용해야 한다. 이렇게 하면 생성자 본문에서 이후에 대입 연산자를 호출하는 오버헤드를 절약할 수 있다. 어떤 경우에는 이것 덕분에 임시 객체의 생성을 방지할 수 있다.

**Chapter3. 가상 함수**

- 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.

**Charter 4. 반환 값 최적화 (Return Value Optimization)**

- 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.

**Chapter 5. 임시 객체(Temporary)**

- 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.

**Chapter 6. 단일 쓰레드 메모리 풀링**

- 유연성은 속도와 절충된다. 메모리 관리의 강력함과 유연성이 높아질수록 실행 속도는 떨어진다.

- 전역 메모리 관리자 (new()와 delete()에 의해 구현되는)는 일반적인 목적을 가지고 있으며, 그러므로 부하를 많이 준다.

- 고정된 크기의 메모리 블록을 주로 할당한다면, 특화된 고정 크기 메모리 관리자가 성능을 현저하게 높여줄 것이다.

- 단일 스레드에 한정된 메모리 블록을 주로 할당한다면 위와 동일한 성능 개선이 가능하다. 단일 스레드 메모리 관리자는 전역 new()와 delete()가 처리해야 하는 동시성을 고려할 필요가 없기 때문에 성능에 많은 도움을 준다.

**Chapter 7. 멀티 스레드 메모리 풀링**

- 전역 메모리 관리자 (new()와 delete()가 구현하는)는 일반적인 목적을 가지고 있기 때문에 부하를 많이 준다.

- 대부분 단일 스레드에 한정된 메모리 블록을 할당한다면 비약적인 성능 개선을 꾀할 수 있다. 단일 스레드 메모리 관리자는 멀티 스레드 메모리 관리자 보다 훨씬 빠르다.

- 효율적인 단일 스레드 할당기를 여러 개 개발한다면, 템플릿을 사용하여 이 할당기들을 멀티스레드 환경으로 쉽게 확장할 수 있다.

**Chapter 8. 인라인의 기본**

- 인라인은 메소드 호출을 메소드의 코드로 교체하는 작업이다.

- 인라인은 호출 오버헤드를 없애고 호출간 최적화가 일어날 수 있도록 하여 성능을 향상 시킨다.

- 인라인은 주로 실행 시간 최적화 이지만, 실행 이미지의 크기도 작게 만들어 준다.

**Chapter 9. 인라인 – 성능 고찰**

- 리터럴 인자와 인라인이 복합될 경우, 컴파일러가 성능을 대폭 개선시킬 수 있는 가능성이 매우 커진다.

- 인라인은 기대에 어긋난 결과를 낳을 수 있고, 너무 과도하게 인라인 작업을 수행하면 반드시 어긋난 결과를 낳게 된다. 인라인은 코드 크기를 증가시킬 수 있다. 코드 크기가 크면 작은 코드에 비해 캐시 누락과 페이지 실패가 자주 발생한다.

- 명백하지 않은 메소드를 인라인으로 만들기 위해서는 단순한 직감이 아닌 샘플 실행 프로파일이 필요하다.

- 정적 크기가 크고 동적 크기가 작은 메소드가 자주 불려진다면, 이 메소드를 다시 작성하여 주요 동적 기능을 밖으로 빼내고 동적 구성 요소를 인라인으로 만들 수 있는지 고려한다.

- 명백한 메소드와 싱글톤 메소드는 항상 인라인이 될 수 있다.

**Chapter 10. 인라인 트릭**

- 인라인은 성능을 개선시킬 수 있다. 목적은 프로그램의 빠른 경로를 찾아 이 경로를 인라인으로 만드는 것이다. 하지만 이 경로를 인라인으로 만드는 것은 명백하지 않을 수 있다.

- 조건부 인라인 작업을 수행하면 메소드가 인라인이 되는 것을 방지 할 수 있다. 컴파일 시간을 줄여주고, 개발 초기의 디버깅을 간략화 시켜준다.

- 지역 정적 병수를 다룰 때에는 조심해야 한다.

- 인라인의 목적은 호출 제거에 있다. 인라인을 사용하기 전에 여러분 시스템의 진정한 호출 부하를 확인하자.

**Chapter 11. 표준 템플릿 라이브러리(Standard Template Library)**

- STL은 추상성, 유연성, 효율성을 모두 갖춘 보기 드문 라이브러리이다.

- 응용프로그램이 어떤 방식으로 컨테이너를 사용하느냐에 따라 가장 효율적인 컨테이너는 달라진다.

- STL이 다루지 못하는 문제 영역에 관한 사항을 알지 못하면, 들인 노력만큼 효율적인 STL 구현의 성능을 이기기는 힘들다.

- 어떤 특정한 시나리오에서 STL 구현의 성능을 능가하는 것은 가능하다.

**Chapter 12. 참조 횟수**

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.

**Chapter 13. 코딩 최적화**

- 코딩 최적화는 국지적 범위를 가지고 있으며, 전체 적인 프로그램 디자인의 이해를 필요로 하지 않는다. 여러분이 아직 디자인을 이해하지 못한 진행 중인 프로젝트에 합류한 경우, 코딩 최적화부터 시작 하면 좋다.

가장 빠른 코드는 절대 실행하지 않는 코드이다. 부하를 많이 주는 연산을 밖으로 빼내기 위해 다음 사항을 점검해 보자.

**1. 결과를 사용할 것인가?** 당연하다는 듯이 들리지만 실제로 일어난다. 종종 연산을 수행해 놓고 결과를 절대 사용하지 않는다.

**2. 결과가 지금 필요한가?** 결과가 실제로 필요한 지점 까지 연산을 연기한다. 미숙한 연산이 어떤 실행 경로에서 절대 사용되지 않을 수 있다.

**3. 결과를 이미 알고 있는가?** 부하를 많이 주는 연산의 결과가 두 줄 위에 나와 있음에도 불구하고 이 연산을 다시 수행하는 것을 보았다. 만약 실행 흐름의 앞부분에서 미리 계산하였다면, 결과를 재사용할 수 있도록 만든다.

가끔 연산을 밖으로 빼낼 수 없으며 수행해야만 하는 경우가 있다. 이제 속도를 증가 시키기 위해 도전

**1. 연산이 전체적으로 일반적인가?** 문에 영역이 필요로 하는 만큼만 유연하면 되고 더 이상은 필요하지 않다. 단순한 가정을 이용하자. 유연함을 감소시키면 속도는 증가한다.

**2. 어떤 유연성은 라이브러리 호출에 숨겨져 있다.** 자주 호출되는 특정 라이브러리 호출을 자체 제작하여 속도를 증가시킬 수 있다. 이 라이브러리들은 여러분의 노력을 정당화 시켜 주기에 충분히 자주 호출되어야 한다. 여러분이 사용하는 이 라이브러리와 시스템 호출의 숨겨진 부하를 대해서 잘 알아야 한다.

**3. 메모리 관리를 최소화 한다.** 대부분의 컴파일러에서 메모리 관리는 상대적으로 부하를 많이 준다.

**4. 모든 입력 데이터의 집합을 고려한다면 이것의 20%가 시간의 80%를 차지한다**. 다른 시나리오를 희생하고 전형적인 입력을 처리하는데 드는 속도를 개선 시킨다.

**5. 캐시, RAM, 디스크 액세스 간의 속도 차이는 상당히 크다**. 캐시 적중이 많이 일어날 만한 코드를 작성한다.