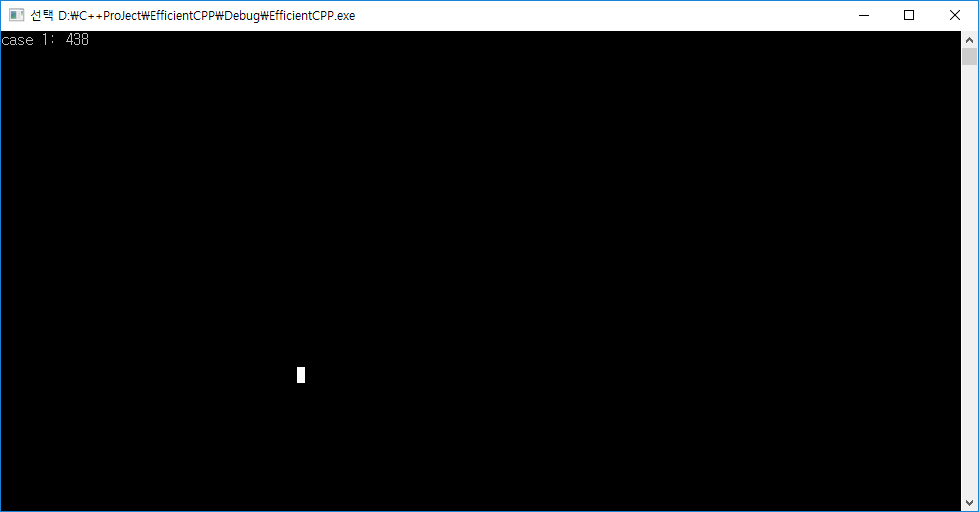
Chapter 2. 생성자와 소멸자

**상속**

상속과 합성은 객체 지향 디자인에서 클래스 들이 서로 연결되는 두 가지 방식이다. 상속 기반 디자인과 생성자/소멸자의 부하 사이의 연결을 살펴보자.

**버전1. pthread\_mutex\_lock()과 pthread\_mutex\_unlock()을 직접 호출 windows mutex를 사용**

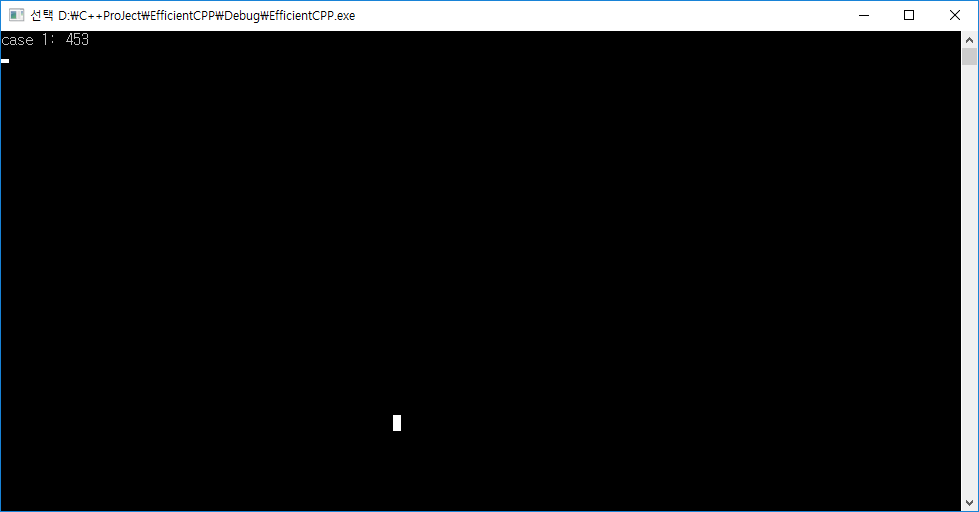
|  |
| --- |
| //버전1.  int main()  {  HANDLE hMutex;  int num = 0;  for (int i = 0; i < 1000000; i++)  {  WaitForSingleObject(hMutex, INFINITE);  num++;  ReleaseMutex(hMutex);  }  return 0;  } |

결과  


**버전2. 기본 클래스로부터 상속되지 않은 독립적인 뮤텍스 객체**

|  |
| --- |
| // 버전2.  class SimpleMutex  {  public:  inline  SimpleMutex::SimpleMutex(HANDLE& mutex) : mutex\_(mutex)  {  WaitForSingleObject(mutex\_, INFINITE);  }  inline  SimpleMutex::~SimpleMutex()  {  ReleaseMutex(mutex\_);  }  private:  HANDLE& mutex\_;  };  int main()  {  HANDLE hMutex;  int num = 0;  hMutex = CreateMutex(NULL, FALSE, NULL);  for (int i = 0; i < 1000000; i++)  {  SimpleMutex m(hMutex);  num++;  }  CloseHandle(hMutex);  return 0;  } |

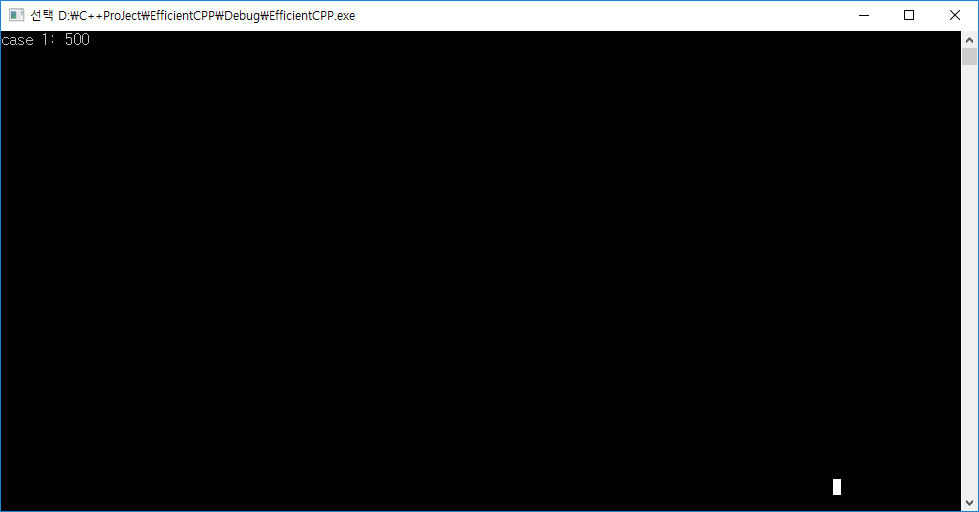
결과



**버전 3. 기본 클래스로부터 파생된 뮤텍스 객체**

|  |
| --- |
| // 버전3.  class BaseMutex  {  public:  BaseMutex(HANDLE& mutex) {};  virtual ~BaseMutex() {};  };  class DerivedMutex : public BaseMutex  {  public:  inline  DerivedMutex::DerivedMutex(HANDLE& mutex) : BaseMutex(mutex), mutex\_(mutex)  {  WaitForSingleObject(mutex\_, INFINITE);  }  inline  DerivedMutex::~DerivedMutex()  {  ReleaseMutex(mutex\_);  }  private:  HANDLE& mutex\_;  };  int main()  {  HANDLE hMutex;  int num = 0;  hMutex = CreateMutex(NULL, FALSE, NULL);  for (int i = 0; i < 1000000; i++)  {  DerivedMutex m(hMutex);  num++;  }  CloseHandle(hMutex);  return 0;  } |

결과



객체의 생성과 소멸은 대게 성능에 나쁜 영향을 준다.

상속 계층에서 하나의 객체를 생성하면 이것의 모든 부모 클래스를 모두 생성한다. 객체 소멸에서도 마찬가지이다.

객체에 연관된 부담은 파생 연결 고리의 복잡도와 길이와도 직접 연결된다.

생성되는 객체의 개수는 파생의 복잡도와 비례한다.

**합성(Composition)**

상속과 같이 객체 합성도 객체 생성과 소멸에 관련된 유사한 성능 문제를 가지고 있다.

객체가 생성, 소멸될 때 포함된 모든 객체들도 생성, 소멸되어야 한다.

|  |
| --- |
| // Trace  class Trace  {  public:  Trace(const char \*name) {};  private:  std::string theFunctionName;  }; |

Trace 객체를 생성하면 std::string 하위 객체도 만들어 진다. Trace 소멸자도 마찬가지 이다.

이것은 자동으로 생성되며 방지할 수 없다.

하위 객체의 생성과 소멸을 좀더 제어하기 위해서는 객체를 포인터로 교체되어야 한다.

|  |
| --- |
| class Trace  {  public:  Trace(const char \*name) {};  private:  std::string \*theFunctionName;  }; |

이제 std::string 객체의 생성과 소멸을 상황에 따라 제어할 수 있게 되었다.

객체의 생성과 소멸을 상황에 따라 제어할 수 있게 되면 불필요한 객체의 생성, 소멸 오버헤드를 줄일 수 있다.

**지연 생성**

C, 파스칼, 다른 유명 언어에서는 데이터 형식을 코드 블록의 시작 부분에 정의해야 한다. 루틴의 시작 부분에서 루틴에서 필요로 하는 모든 변수를 정의한다.

C, C++의 차이점으로 C++ 언어의 시초부터 C++ 언어 전문가는 변수를 처음 사용할 때까지 생성하지 않는 방법을 주장해왔다. 때문에 C++에서 진부한 C 선언 문법을 여전히 사용하면 성능에 많은 영향을 줄 수 있다.

C++이 C보다 나은 점 중 하나는 변수 생성을 지연시킬 수 있다는 점이다.

때문에 C++ 프로그래머는 객체 정의를 어디에서 해야 할지에 주의해야 한다.

**중복 생성**

포함된 객체의 중복 생성으로 오버헤드를 발생시킬 수 있다.

|  |
| --- |
| class Person  {  public:  Person(const char \*s) { name = s; }  private:  std::string name;  }; |

위 코드에서 Person 객체는 Person 생성자의 본문이 실행되기 전에 초기화되어야 한다.

포함된 string 객체는 Person 생성자 초기화 리스트에 없다.

그러므로 컴파일러는 string 멤버 객체를 초기화 하기 위해 string 기본 생성자를 호출한다.

또 name = s; 구문을 위해 string::operator=(const char\*)를 호출해야 한다.

대입 연산 덕분에 먼저 호출된 string의 기본 생성자는 아무 의미가 없어졌다.

이 해결을 위해서는 string 멤버 name을 명시적으로 초기화 하면 된다.

|  |
| --- |
| Person(const char \*s) : name(s) { } |

**키 포인트**

- 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.