Chapter 11. 표준 템플릿 라이브러리(Standard Template Library)

- 표준 템플릿 라이브러리는 컨테이너와 일반 알고리즘의 강력한 조합

**후방 삽입**

- 삽입 실험에서 무작위의 백 만개요소를 배열, 벡터, 리스트, 멀티셋에 삽입한다.

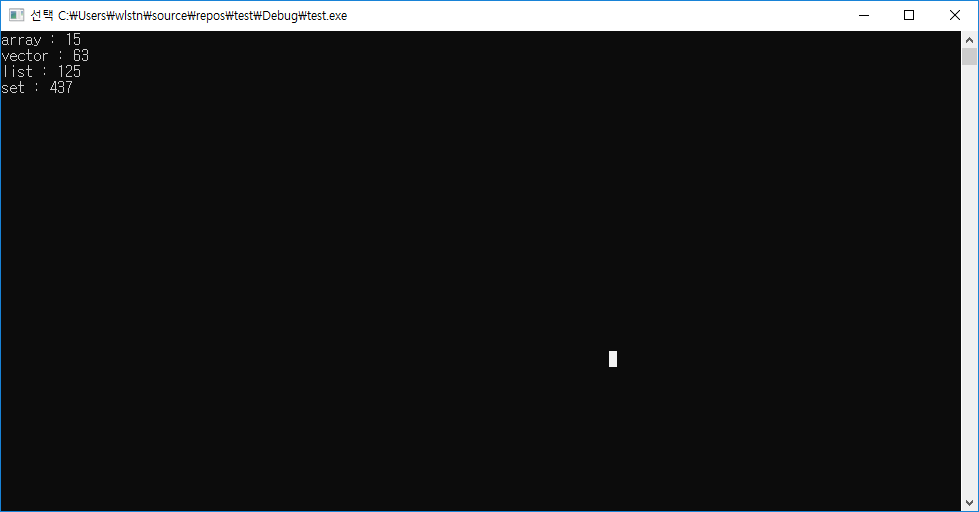
- 테스트 대상 컨테이너에 대한 포인터

- 삽입할 요소들의 데이터 배열에 대한 포인터

- 데이터 배열의 크기

|  |
| --- |
| void arrayinsert()  {  std::array<int, 100000> array\_;  for (int k = 0; k < 100000; ++k)  array\_[k] = rand()% 100000;  }  void vectorInsert()  {  std::vector<int> v;  for (int k = 0; k < 100000; ++k)  v.push\_back(rand() % 100000);  }  void listInsert()  {  std::list<int> l;  for (int k = 0; k < 100000; ++k)  l.push\_back(rand() % 100000);  }  void multisetInsert()  {  std::set<int> s;  for (int k = 0; k < 100000; ++k)  s.insert(rand() % 100000);  }  int main()  {  unsigned long long s = ::GetTickCount64();  arrayinsert();  unsigned long long time = ::GetTickCount64() - s;  std::cout << "array : " << time << std::endl;  s = ::GetTickCount64();  vectorInsert();  time = ::GetTickCount64() - s;  std::cout << "vector : " << time << std::endl;  s = ::GetTickCount64();  listInsert();  time = ::GetTickCount64() - s;  std::cout << "list : " << time << std::endl;  s = ::GetTickCount64();  multisetInsert();  time = ::GetTickCount64() - s;  std::cout << "set : " << time << std::endl;  return 0;  } |

실행결과



- 배열은 다른 컨테이너에 비해 한가지 커다란 장점을 가지고 있다, 처음부터 모든 데이터 요소를 포함하고 있을 수 있다.

- 벡터, 리스트, 셋은 컬렉션이 얼마나 커질지 미리 알지 못한다.

- 리스트는 데이터를 저장하기 위한 리스트 요소를 할당 해야 할 뿐만 아니라, 전방/후방 포인터를 설정해야 한다.

- 셋은 항상 컬렉션이 정렬된 순서가 되도록 유지해야 한다.

- 배열과 벡터는 연속된 메모리 블록을 차지하는 시퀀스 컨테이너이다.

- **컬렉션이 앞으로 얼마나 커질지 모른다면 벡터 컨테이너**가 더욱 도움이 될 것이다.

- 벡터의 크기는 벡터가 현재 포함하고 있는 요소의 개수, 용량은 벡터가 증분을 수용할 수 있도록 추가적인 메모리를 할당하지 않아도 되는 한도 에서 저장할 수 있는 최대 개수

**벡터의 삽입**

- 벡터에 첫 번째 요소를 삽입하면, 메모리의 커다란 블록을 할당하여 **초기 크기(1)보다 큰 벡터 용량을 설정**

- 이후 계속 커진다면, 결국 벡터 크기는 용량과 같아지게 되며, 다음 삽입 시에는 벡터 구현상 자신의 용량을 확장

**벡터의 확장**

- 여분의 요소들을 위한 공간을 만들기 위해 더욱 큰 메모리 블록을 할당

- 기존 컬렉션 요소들을 새롭게 할당한 메모리에 복사한다. 예전 컬렉션의 각 요소에 대해 복사 생성자를 호출

- 예전 컬렉션을 소멸하고 메모리를 해지한다. 예전 컬렉션 복사본의 각 요소에 대해 소멸자를 호출한다.

- 벡터의 확장은 **부하를 많이 줄 수 있다.**

**벡터 확장의 오버헤드 대처**

- **확장을 자주 하게 되는 경우 list로 대체하는 것**이 더 빠를 수 있다.

- 객체에 대한 vector 선언 보단 객체의 포인터에 대한 vector 선언을 사용한다.

- 벡터에서 사용할 용량을 미리 예약해 놓을 수 있다**(reserve(n))**

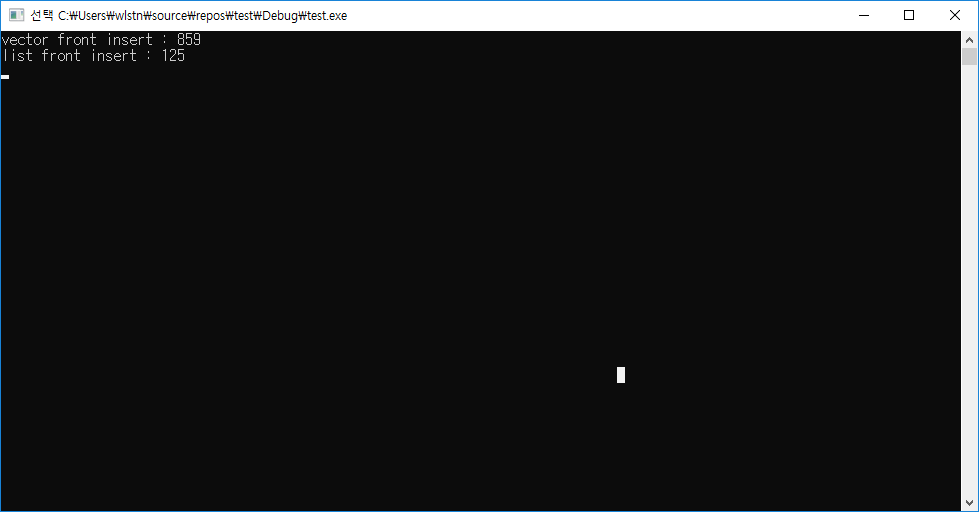
|  |
| --- |
| std::vector<int> v;  v.reserve(100000); |

**전방 삽입**

- 이전 까진 요소들의 맨 뒤에 삽입을 했지만, 앞에 삽입하는 경우는 이야기가 달라 질 수 있다.

|  |
| --- |
| void vectorFrontInsert()  {  std::vector<int> v;  for (int k = 0; k < 100000; ++k)  v.insert(v.begin(), rand() % 100000);  }  void listFrontInsert()  {  std::list<int> l;  for (int k = 0; k < 100000; ++k)  l.push\_front(rand() % 100000);  }  int main()  {  unsigned long long s = ::GetTickCount64();  vectorFrontInsert();  unsigned long long time = ::GetTickCount64() - s;  std::cout << "vector front insert : " << time << std::endl;  s = ::GetTickCount64();  listFrontInsert();  time = ::GetTickCount64() - s;  std::cout << "list front insert : " << time << std::endl;  return 0;  } |

실행 결과



- STL 디자인은 전방 삽입할 경우 벡터를 사용할 것을 미리 예방차원에서 push\_front() 메소드를 벡터에서 지원하지 않음

**삭제**

- 벡터는 요소의 후방 삽입(삭제)에서 좋은 성능을 낸다. 이것은 컬렉션 크기와 무관한 상수 시간 연산

- 후방이 아닌 다른 곳에 요소를 삽입(삭제)하고자 할 때 벡터는 최악의 선택이다. 이러한 삽입(삭제)의 부하는 벡터의 삽입(삭제) 지점과 마지막 요소 사이의 거리에 따라 비례한다.

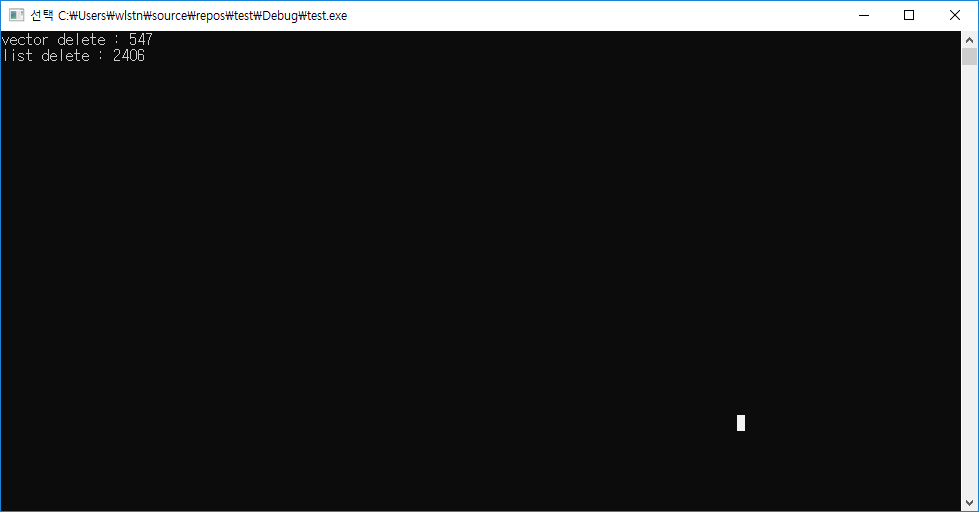
- 큐는 컬렉션의 전방과 후방에 사입(삭제)을 수행하는 데 효율 적이다. (상수 시간), 그 외에 다른 곳에 삽입(삭제)를 수행하는 것은 비효율 적이다.

- 리스트는 컬렉션 어느 곳이라도 삽입(삭제)을 수행하는 데 효율적이다.(상수 시간)

**후방 삭제**

|  |
| --- |
| std::vector<int> v\_;  std::list<int> l\_;  void vectorDelete()  {  while (v\_.empty() == false)  v\_.pop\_back();  }  void listDelete()  {  while (l\_.empty() == false)  l\_.pop\_back();  }  int main()  {  for (int k = 0; k < 100000; ++k)  {  v\_.push\_back(k);  l\_.push\_back(k);  }  unsigned long long s = ::GetTickCount64();  vectorDelete();  unsigned long long time = ::GetTickCount64() - s;  std::cout << "vector delete : " << time << std::endl;  s = ::GetTickCount64();  listDelete();  time = ::GetTickCount64() - s;  std::cout << "list delete : " << time << std::endl;  return 0;  } |

실행 결과

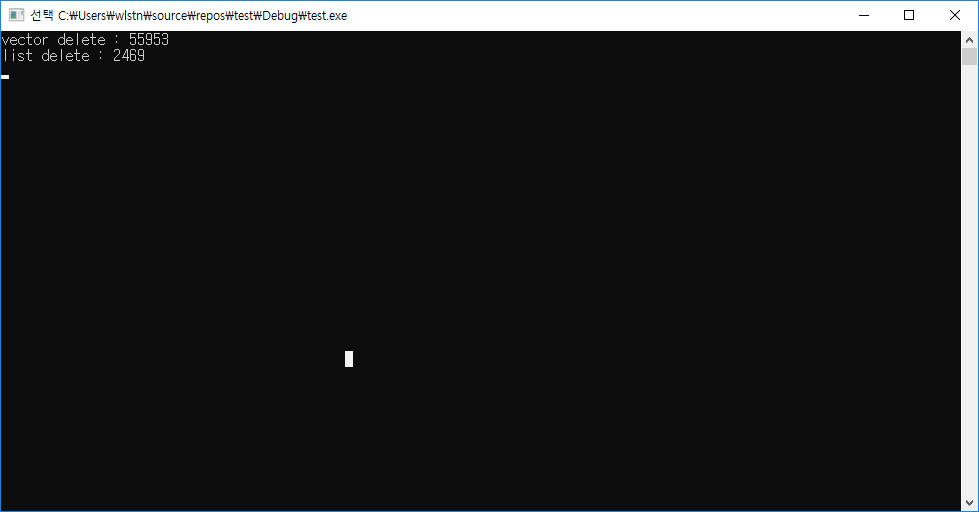


- 삽입과 같이 후방에서 요소를 삭제할 때 벡터 컨테이너는 리스트보다 성능이 우수하다.

**전방 삭제**

|  |
| --- |
| std::vector<int> v\_;  std::list<int> l\_;  void vectorFrontDelete()  {  while (v\_.empty() == false)  v\_.erase(v\_.begin());  }  void listFrontDelete()  {  while (l\_.empty() == false)  l\_.pop\_front();  }  int main()  {  for (int k = 0; k < 1000000; ++k)  {  v\_.push\_back(k);  l\_.push\_back(k);  }  unsigned long long s = ::GetTickCount64();  vectorFrontDelete();  unsigned long long time = ::GetTickCount64() - s;  std::cout << "vector delete : " << time << std::endl;  s = ::GetTickCount64();  listFrontDelete();  time = ::GetTickCount64() - s;  std::cout << "list delete : " << time << std::endl;  return 0;  } |

실행 결과



- 전방 삽입과 같이 벡터의 전방 삭제는 비효율적

**통과**

- 컨테이너의 요소들을 한 번에 통과하면서 각 요소에 대해 연산을 수행해야 하는 경우가 존재

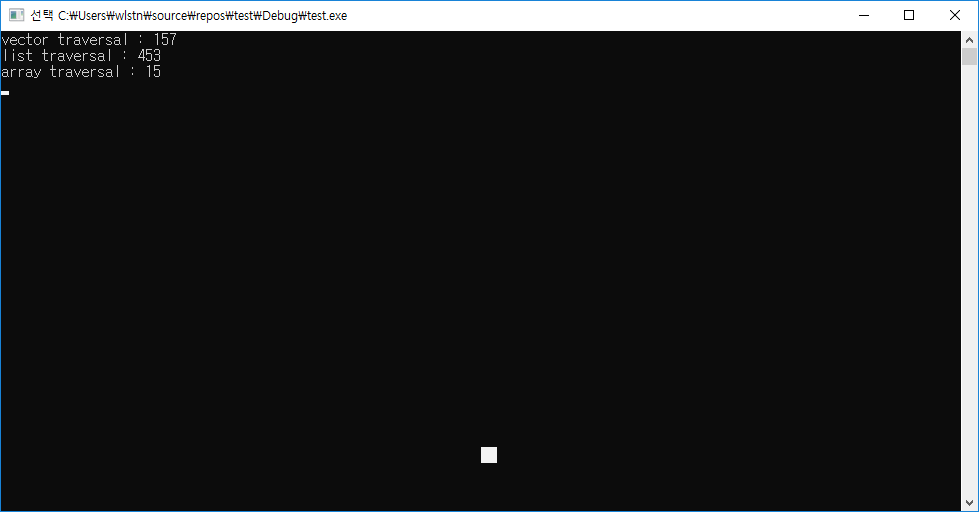
- 컨테이너 통과를 표현하기 위해 STL의 accumulate() 함수를 구현

- accumulate 함수는 컨테이너의 처음부터 끝까지 통과하면서 모든 요소를 더한다.

- 테스트를 수행한 컨테이너는 배열, 벡터, 리스트 이다.

|  |
| --- |
| std::vector<int> v\_;  std::list<int> l\_;  std::array<int, 1000000> a\_;  void vectorTraverse()  {  int a;  for (int i = 0; i < 1000000; ++i)  a = v\_[i];}  void listTraverse()  {  int a;  for (std::list<int>::iterator itor = l\_.begin(); itor != l\_.end(); ++itor)  a = \*itor;  }  void arrayTraverse()  {  int a;  for (int i = 0 ; i < 1000000 ; ++i )  a = a\_[i];  }  int main()  {  for (int k = 0; k < 1000000; ++k)  {  v\_.push\_back(k);  l\_.push\_back(k);  a\_[k] = k;  }  unsigned long long s = ::GetTickCount64();  vectorTraverse();  unsigned long long time = ::GetTickCount64() - s;  std::cout << "vector traversal : " << time << std::endl;  s = ::GetTickCount64();  listTraverse();  time = ::GetTickCount64() - s;  std::cout << "list traversal : " << time << std::endl;  s = ::GetTickCount64();  arrayTraverse();  time = ::GetTickCount64() - s;  std::cout << "array traversal : " << time << std::endl;  return 0;  } |

실행 결과



- 벡터와 배열 컨테이너는 리스트 보다 훨씬 우수 하다.

- 벡터와 배열은 모두 자신의 컬렉션을 연속적인 메모리에 저장하고 있다. 즉 인접한 컬렉션 요소는 물리적으로도 메모리상 가깝다는 것이다.

- 리스트는 근접한 요소들은 메모리 상 근접해 있지 않다. 하물며 리스트는 요소값 이외에 전방/후방 포인터를 저장하고 있어야 하기 때문에 해당 벡터 요소보다 크기가 더 크다.

- 크기가 크기 때문에 캐시선에 로드되는 요소들은 몇 개 없을 것이다.

**찾기**

- std::find 함수를 사용하는 것은 비효율 적이다.

- set의 경우 자체 find()함수를 지원하는 경우 그 함수를 사용하라 훨씬 효율적일 것이다.

**함수 객체**

- 함수 객체를 사용한 버전은 함수 포인터를 사용한 것 보다 훨씬 우수한 성능을 가지고 있다.

- 함수 포인터는 실행될 때 까지 확인될 수 없기 때문에 인라인으로 만들어 질 수 없다.

- 하지만 함수 객체는 컴파일 시에 결정되기 때문에, 컴파일러는 operator() 호출을 인라인으로 만들 수 있으며, 성능을 증가 시킬 수 있다.

**키 포인트**

- 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.