**컨테이너**

컨테이너 : 동질적인 객체의 집합을 저장하고 관리하는 역할

① 시퀀스 컨테이너(Sequence Container) : 자료의 선형적인 집합이며 자료를 저장하는 기본 임무에 충실한 가장 일반적인 컨테이너이다. 삽입된 자료를 무조건 저장하며 입력되는 자료에 특별한 제약이나 관리 규칙은 없다. 사용자는 시퀀스의 임의 위치에 원하는 요소를 마음대로 삽입, 삭제할 수 있다. STL에는 벡터, 리스트, 데크 세 가지의 시퀀스 컨테이너가 제공된다.

② 연관 컨테이너(Associative Container) : 자료를 무조건 저장하기만 하는 것이 아니라 일정한 규칙에 따라 자료를 조직화하여 관리하는 컨테이너이다. 정렬이나 해시 등의 방법을 통해 삽입되는 자료를 항상 일정한 기준(오름차순, 해시 함수)에 맞는 위치에 저장해 놓으므로 검색 속도가 빠른 것이 장점이다. 표준 STL에는 정렬 연관 컨테이너인 셋, 맵 등의 컨테이너가 제공된다.

③ 어댑터 컨테이너 (Adapter Container) : 시퀀스 컨테이너를 변형하여 자료를 미리 정해진 일정한 방식에 따라 관리하는 것이 특징이다. 스택, 큐, 우선 순위 큐 세 가지가 있는데 스택은 항상 LIFO의 원리로 동작하며 큐는 항상 FIFO의 원리로 동작한다. 자료를 넣고 빼는 순서를 외부에서 마음대로 조작할 수 없으며 컨테이너의 규칙대로 조작해야 한다.

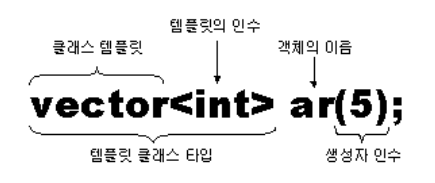
* 시퀀스는 삽입 , 삭제에 별 제약이 없다
* 연관은 신속한 검색을 위해 찾기 좋은 위치에 자동 삽입
* 어댑터는 FIFO , LIFO 등 미리 정한 규칙의 통제를 따른다.

벡터

* 요소의 개수에 맞게 자동으로 메모리를 재 할당하여 크기를 신축적으로 늘릴 수 있는 배열

(쉽게말하면 동적배열)

* 단순한 동적 배열에 비해 템플릿 기반이므로 요소 타입에 무관한 배열을 만들 수 있는 것이 큰 장점



🡪 ar은 지금 선언하고자 하는 객체, 더 쉽게 표현하면 변수의 이름

🡪 (5)는 이 클래스의 생성자로 전달되는 인수

🡪 생성자의 인수로 크기를 생략하면 요소를 가지지 않는 빈 벡터가 만들어진다

🡪 벡터는 동적으로 메모리를 관리하므로 실행중에 요소를 얼마든지 더 추가할 수 있다

vector 예제

#include <iostream>

#include <vector>

using namespace std;

int main()

{

int num;

int i;

printf("배열 크기를 입력하시오 : ");

cin>>num;

vector<int> vi(num);

for (i=0;i<num;i++) {

vi[i]=i\*2;

}

for (i=0;i<num;i++) {

printf("vi[%d]=%d\n",i,vi[i]);

}

printf("벡터의 크기는 %d입니다.\n",vi.size());

}

* STL의 구성 요소들은 전부 헤더 파일에 선언되어 있다. 그래서 사용하고자 하는 구성 요소를 정의하는 헤더 파일을 반드시 include해야 한다 (vector 사용하기위해 #include <vector> 사용)
* 벡터의 할당 크기를 조사하려면 size 멤버 함수를 호출한다. 정적 배열은 일단 선언되면 실행 중에 그 크기를 알 수 없지만 벡터는 내부에서 크기를 관리하므로 size 함수로 현재 크기를 정확하게 조사할 수 있다

push\_back 예제

#include <iostream>

#include <vector>

using namespace std;

void main()

{

int i;

vector<int> vi;

for (i=0;i<10;i++) {

vi.push\_back(i\*2);

}

for (i=0;i<10;i++) {

printf("vi[%d]=%d\n",i,vi[i]);

}

printf("벡터의 크기는 %d입니다.\n",vi.size());

}

* push\_back은 제일 뒤에 새로운 요소 하나를 추가하며 이 과정에서 벡터의 메모리가 부족하다면 재할당된다

리스트 : 이중 연결 리스트로 구현된 컨테이너

🡪리스트의 요소들은 ‘노드’ 라는 구조체로 관리되며 노드끼리는 링크로 서로 연결되있어 요소의 논리적인 순서를 기억한다

🡪링크에 의해 연결 될 뿐 이므로 인접한 메모리에 배치되지 않아도 상관없으며 삽입, 삭제할 때도 앞뒤 노드의 링크만 조작하므로 대용량의 메모리를 밀고 당길 필요가 없다.

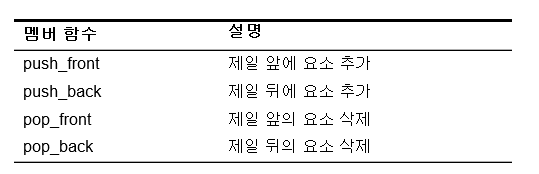
🡪삽입 , 삭제를 할 때도 앞뒤 노드의 링크만 조작하므로 대용량 메모리를 밀고 당길 필요가 없다. 그래서 속도가 빠르다.

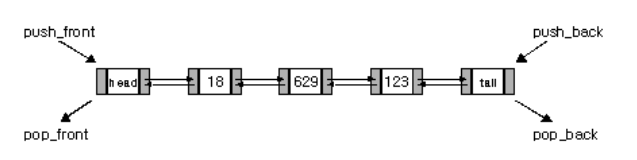
🡪반면 리스트의 한 요소를 찾으려면 첫 노드부터 순서대로 링크를 따라 이동해야 하므로 읽기 속도는 무척 느리다.

* 동일한 자료의 집합을 관리한다는 면에서 벡터와 용도는 동일하지만 내부적인 구조가 상이해서 확연히 다른 특징을 가진다.
* 리스트는 사입, 삭제에 강한 컨테이너다 / 벡터는 읽고 쓰기에 간한 컨테이너다

list<T> Name;

* 리스트는 최초 빈 상태로 만들어도 빠른 속도로 삽입, 삭제를 할 수 있으므로 통상 빈 상태로 생성한다





list 예제

#include <iostream>

#include <list>

using namespace std;

void main()

{

list<int> li;

int i;

for (i=0;i<5;i++) {

li.push\_back(i\*2);

}

list<int>::iterator it;

for (it=li.begin(),i=0;it!=li.end();it++,i++) {

printf("%d번째=%d\n",i,\*it);

}

}

맵 : 두개씩 짝을 이루는 데이터를 저장하는 컨테이너

* 연관 컨테이너의 일종인 맵은 시퀀스와 달리 아무렇게나 요소를 저자하지 않으며 정렬된 위치에 삽입한다.
* 항상 정렬된 상태로 관리되므로 이분 검색 기법을 사용할 수 있으며 따라서 요소가 아무리 많아도 굉장히 빠르게 검색 할 수 있다.
* 대량의 데이터를 신속하게 검색해야 할 필요가 있을 때 맵이 주로 사용

map 예제

#include <iostream>

#include <string>

#include <map>

using namespace std;

struct SProduct

{

string Name;

int Price;

} arPro[]={

{"맛동산",500},{"박카스",400},{"네스카페",250},{"신라면",450},

{"88라이트",1900},{"불티나",300},{"스타킹",700},{"김치",2000},

{"신문",500},{"비타500",500},{"비타1000",1000},{"왕꿈틀이",900},

{"뽀빠이",200},{"위스퍼",800},{"콘텍600",600},{"페리오치약",2200},

{"모나미볼펜",90},{"까페라떼",990},{"밧데리",1000},{"쵸코파이",250},

};

void main()

{

map<string,int> mPro;

map<string,int>::iterator it;

int i;

string Name;

for (i=0;i<sizeof(arPro)/sizeof(arPro[0]);i++) {

mPro[arPro[i].Name]=arPro[i].Price;

}

for (;;) {

cout<< "상품명을 입력하시오(끝낼때는 '끝'입력) : ";

cin>> Name;

if (Name=="끝") break;

it=mPro.find(Name);

if (it == mPro.end()) {

cout<< "그런 제품은 없습니다." << endl;

} else {

cout<< Name << "의 가격은 " << it->second << "입니다." << endl;

}

}

}