표준 템플릿 라이브러리 (STL)

(4주차)

학습개요

- 학습 목표
  - C++ 표준 템플릿 라이브러리의 동작 원리와 구현 원리를 이해할 수 있다.
  - C++ 표준 템플릿 라이브러리가 제공하는 기능들을 활용할 수 있다.

- 학습 내용
  - 표준 템플릿 라이브러리
  - 컨테이너
  - 반복자
  - 알고리즘
  - 함수객체
  - 어댑터
  - 할당기
  - 실습

STL: Standard Template Library

- 프로그램에 필요한 자료구조와 알고리즘을 템플릿으로 제공하는 표준 라이브러리
- 자료구조와 알고리즘은 반복자라는 구성요소를 통해 연결
- STL 구성요소
  - 컨테이너(container): 객체를 저장하는 컬렉션 객체
  - 반복자(iterator): 포인터와 유사한 개념으로 컨테이너의 원소를 가리키고, 순회하는 기능을 제공
  - 알고리즘(Algorithm): 정렬, 삭제, 검색, 연산 등을 해결하는 일반화된 방법을 제공하는 함수 템플릿
  - 함수 객체(Function Object): 함수처럼 동작하는 객체로 operator() 연산자를 오버로딩한 객체
  - 어댑터(Adaptor): 구성요소의 인터페이스를 변경해 새로운 인터페이스를 갖는 구성요소로 변경
  - 할당기(Allocator): 컨테이너의 메모리 할당 정책을 캡슐화한 객체

컨테이너(container)

- 같은 타입의 객체를 저장하고 관리할 목적으로 제공되는 클래스
- 시퀀스(sequence) 컨테이너: 컨테이너의 원소가 삽입된 위치 순서를 갖는 선형 컨테이너 – vector, deque, list
- 연관(associative) 컨테이너: 컨테이너의 원소가 삽입 순서와 다르게 자동 정렬되는 비선형 컨테이너 – map, multimap, set, multiset
- 배열 기반 컨테이너: vector, deque
- 노드 기반 컨테이너: list, map, multimap, set, multiset

vector

- 시퀀스 컨테이너이며, 배열 기반의 컨테이너
- 컨테이너 끝에서 데이터를 추가하고 삭제하기 위한 push_back()과 pop()_back() 함수를 제공
- operator[] 연산자 오버로딩을 통해 일반 배열처럼 컨테이너 원소에 접근 가능
- 원소의 갯수를 반환하는 size() 함수를 제공

```
#include <iostream>
#include <vector>
using namespace std;

void main( )
{
    vector<int> v; // 정수를 저장하는 컨테이너 v 생성

    v.push_back(10); // v에 정수 추가
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for(unsigned int i = 0 ; i < v.size() ; ++i)
        cout << v[i] << endl; // v[i]는 i번째 index의 정수 반환
}
```

결과

```
10
20
30
40
50
```

실습 – 컨테이너

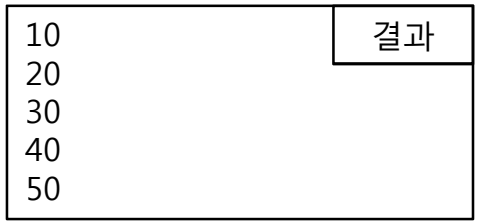
반복자(iterator) (1)

- 포인터와 비슷하게 동작 (* 연산자 제공)
- 컨테이너에 저장된 원소를 순회하고 접근하는 일반화된 방법을 제공 (++, !=, == 연산자 제공)
- 컨테이너와 알고리즘이 하나로 동작하게 묶어주는 인터페이스 역할
- 모든 컨테이너는 자신만의 반복자를 제공
- begin()과 end() 가 순차열의 처음과 끝을 가리키는 반복자를 반환. 이 때 끝은 마지막 원소 다음을 가리킴

```
vector<int> v;

v.push_back(10);
v.push_back(20);
v.push_back(30);
v.push_back(40);
v.push_back(50);

vector<int>::iterator iter; // 반복자 생성(아직 원소를 가리키지 않음)
for(iter = v.begin() ; iter != v.end() ; ++iter)
    cout << *iter << endl; // 반복자가 가리키는 원소를 역참조
```



반복자(iterator) (2)

```
vector<int> v;

v.push_back(10);
v.push_back(20);
v.push_back(30);
v.push_back(40);
v.push_back(50);

vector<int>::iterator iter=v.begin(); //시작 원소를 가리키는 반복자
cout << iter[0] << endl; // [] 연산자
cout << iter[1] << endl;
cout << iter[2] << endl;
cout << iter[3] << endl;
cout << iter[4] << endl;
cout << endl;

iter += 2; // += 연산
cout << *iter << endl;
cout << endl;

vector<int>::iterator iter2 = iter+2; // + 연산
cout << *iter2 << endl;
```

결과

```
10
20
30
40
50

30

50
```

실습 – 반복자

알고리즘 (1)

- 순차열의 조사, 변경, 관리, 처리를 목적으로 제공
- STL 컨테이너에 저장된 원소들을 대상으로 하는 함수 템플릿
- 대부분의 알고리즘은 algorithm 헤더 파일에 정의되어 있고, 수치 관련 알고리즘은 numeric 헤더 파일에 정의되어 있음
- copy, find, sort, unique 등 유용한 알고리즘이 구현되어 있음

```
#include <algorithm> // find 사용
...
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator iter;
    iter = find(v.begin(), v.end(), 20); //[begin, end)에서 20 찾기
    cout << *iter << endl;

    iter = find(v.begin(), v.end(), 100); //[begin, end)에서 100 찾기
    if( iter == v.end() ) // 100이 없으면 iter==v.end() 임
        cout << "100이 없음!" << endl;
```

결과
20 100이 없음

알고리즘 (2)

```
#include <vector>
#include <list>
#include <algorithm>

...

    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    list<int> lt;
    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(40);
    lt.push_back(50);

    sort(v.begin(), v.end()); // 정렬 가능(vector는 임의 접근 반복자)
    sort(lt.begin(), lt.end()); // 에러!(list는 양방향 반복자)
```

실습 – 알고리즘

함수객체

- 클라이언트가 정의한 동작을 다른 구성 요소에 반영하려고 할 때 사용
- STL 알고리즘이 함수 객체, 함수, 함수 포인터 등의 함수류를 인자로 받아들여 알고리즘을 유연하게 동작시킴

```
#include <vector>
#include <algorithm>
...

    vector<int> v;
    v.push_back(50);
    v.push_back(10);
    v.push_back(20);
    v.push_back(40);
    v.push_back(30);

    sort(v.begin(), v.end(), less<int>() ); // 오름차순 정렬
    for(vector<int>::iterator iter= v.begin() ; iter != v.end() ; ++iter)
        cout << *iter << " ";
    cout << endl;

    sort(v.begin(), v.end(), greater<int>() ); // 내림차순 정렬
    for(vector<int>::iterator iter= v.begin() ; iter != v.end() ; ++iter)
        cout << *iter << " ";
    cout << endl;
```

결과

```
10 20 30 40 50
50 40 30 20 10
```

실습 – 함수객체

어댑터 (1)

- 어댑터는 구성요소의 인터페이스를 변경
- 컨테이너 어댑터, 반복자 어댑터, 함수 어댑터
- stack 컨테이너 어댑터는 시퀀스 컨테이너를 LIFO(Last-In First-Out) 방식의 스택 컨테이너로 변환 (디폴트 컨테이너는 deque)
- reverse_iterator 반복자 어댑터는 반복자의 동작 방식을 반대로 동작시키는 역방향 반복자로 변환
- not2 함수 어댑터는 함수 객체를 NOT(반대)로 변환

```
#include <stack>
...
    stack<int> st; //stack 컨테이너 생성
    st.push( 10 ); // 데이터 추가(입력)
    st.push( 20 );
    st.push( 30 );
    cout << st.top() << endl; // top 데이터 출력
    st.pop(); // top 데이터 삭제
    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();

    if( st.empty() ) // 스택이 비었는지 확인
        cout << "stack에 데이터 없음" << endl;
```

결과

```
30
20
10
statck에 데이터 없음
```

어댑터 (2)

```
#include <vector>
#include <stack>
...
    stack<int, vector<int> > st; // vector 컨테이너를 이용한 stack 컨테이너 생성

    st.push( 10 ); // 데이터 추가(입력)
    st.push( 20 );
    st.push( 30 );

    cout << st.top() << endl; // top 데이터 출력
    st.pop(); // top 데이터 삭제
    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();

    if( st.empty() ) // 스택이 비었는지 확인
        cout << "stack이 데이터 없음" << endl;
```

결과

```
30
20
10
statck에 데이터 없음
```

어댑터 (3)

```
#include <vector>
...
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for( vector<int>::iterator iter= v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    //일반 반복자 iterator를 역방향 반복자 reverse_iterator로 변환
    reverse_iterator< vector<int>::iterator > riter(v.end());
    reverse_iterator< vector<int>::iterator > end_riter(v.begin());

    for(   ; riter != end_riter ; ++riter )
        cout << *riter << " ";
    cout << endl;
```

결과

```
10 20 30 40 50
50 40 30 20 10
```

어댑터 (4)

```
#include <vector>
...
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    for( vector<int>::iterator iter= v.begin(); iter != v.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    // STL 모든 컨테이너는 반복자 어댑터 reverse_iterator를 typedef 타입으로 정의하며
    // rbegin(), rend()로 컨테이너의 역방향 반복자를 반환함.
    vector<int>::reverse_iterator riter(v.rbegin());
    for(   ; riter != v.rend() ; ++riter )
        cout << *riter << " ";
    cout << endl;
```

결과

```
10 20 30 40 50
50 40 30 20 10
```

어댑터 (5)

```
#include <functional> //not2 사용
...
    cout << less<int>()(10, 20) << endl; //임시 less 객체로 비교
    cout << less<int>()(20, 20) << endl;
    cout << less<int>()(20, 10) << endl;
    cout << "==============" <<endl;
    cout << not2( less<int>() )(10, 20) << endl;// 임시 객체 less에 not2 어댑터 적용
    cout << not2( less<int>() )(20, 20) << endl;
    cout << not2( less<int>() )(20, 10) << endl;
    cout << endl;

    less<int> l;
    cout << l(10, 20) << endl; // less 객체 l로 비교
    cout << l(20, 20) << endl;
    cout << l(20, 10) << endl;
    cout << "==============" <<endl;
    cout << not2( l )(10, 20) << endl; // less 객체 l에 not2 어댑터 적용
    cout << not2( l )(20, 20) << endl;
    cout << not2( l )(20, 10) << endl;
```

결과

```
1
0
0
==============
0
1
1

1
0
0
==============
0
1
1
```

실습 – 어댑터

할당기

- 컨테이너의 메모리 할당 정보와 정책(메모리 할당 모델)을 캡슐화한 STL 구성요소
- 할당기는 템플릿 클래스
- 모든 컨테이너는 템플릿 매개변수에 할당기를 인자로 받음
- 기본 할당기는 allocator<T>
- 컨테이너는 템플릿 매개변수에 디폴트 매개변수 값으로 기본 할당기를 지정

```
#include <vector>
#include <set>
...
    // vector<typename T, typename Alloc = allocator<T> >
    // vector<int> 와 같음
    vector< int, allocator<int> > v;
    v.push_back( 10 );
    cout << *v.begin() << endl;

    // set<typename T, typename Pred = less< T >, typename Alloc = allocator<T> >
    // set<int> 와 같음
    set< int, less<int>, allocator<int> > s;
    s.insert( 10 );
    cout << *s.begin() << endl;
```

결과
10 10

실습 – 할당기

학습정리

- STL은 프로그램에 필요한 자료구조와 알고리즘을 템플릿으로 제공하고, 자료구조와 알고리즘은 반복자라는 구성요소를 통해 연결된다.
- STL 구성요소인 컨테이너는 객체를 저장하는 컬렉션 객체로, 시퀀스 컨테이너(vector, deque, list)와 연관 컨테이너(map, multimap, set, multiset)를 제공한다. 이 중 vector, deque는 배열 기반 컨테이너이며, list, map, multimap, set, multiset는 노드 기반 컨테이너이다.
- STL 구성요소인 반복자는 컨테이너의 원소를 가리키고, 순회하는 기능을 제공하며, 컨테이너마다 고유의 반복자를 가진다.
- STL 구성요소인 알고리즘은 정렬, 삭제, 검색, 연산 등을 해결하는 일반화된 방법을 제공하는 함수 템플릿이다.
- STL 구성요소인 함수 객체는 함수처럼 동작하는 객체로 operator() 연산자를 오버로딩한 객체이다.
- STL 구성요소인 어댑터는 구성요소의 인터페이스를 변경해 새로운 인터페이스를 갖는 구성요소로 변경할 수 있도록 해준다.
- STL 구성요소인 할당기는 컨테이너의 메모리 할당 정책을 캡슐화한 객체이다.