객체지향프로그래밍

Lecture 13 : 템플릿과 STL

충북대 소프트웨어학부

이 태 겸(showm321@gmail.com)

목차

❖ 클래스 템플릿

❖ STL

템플릿

❖ 템플릿(template): 틀

❖ C++에서 템플릿: 코드의 틀, 타입의 일반화

❖ 함수 템플릿, 클래스 템플릿

- C++의 템플릿을 이용하면 함수나 클래스를 정의할 때 특정 데이터 형을 사용하는 대신 범용형을 사용할 수 있다.
- 함수 템플릿이나 클래스 템플릿은 여러가지 데이터 형에 대해서 함수 정의나 클래스 정의를 생성할 수 있다.

**템플릿은 처리 알고리즘은 동일하고,
처리할 값의 데이터 형이 다양할 때 유용하게 사용**

템플릿 장점과 제네릭 프로그래밍

❖ 템플릿 장점
- 함수 코드의 재사용
  - 높은 소프트웨어의 생산성과 유용성

❖ 템플릿 단점
- 포팅에 취약
  - 컴파일러에 따라 지원하지 않을 수 있음
- 컴파일 오류 메시지 빈약, 디버깅에 많은 어려움
  - 에러 메시지가 매우 길고 복잡함

❖ 제네릭 프로그래밍
- generic programming
  - 일반화 프로그래밍이라고도 부름, 하나의 코드로 다양한 상황이나 타입을 처리할 수 있는 프로그래밍 방법
  - 템플릿을 활용해 함수와 클래스를 구현하는 프로그래밍 기법
  - C++에서 STL(Standard Template Library) 제공 및 활용
- 보편화 추세
  - Java, C# 등 많은 언어에서 활용

클래스 템플릿의 선언

- ❖ 멤버들의 타입이 다른 클래스 필요
- ❖ 제네릭 클래스 만들기

```
class I_Queue
{
    // 멤버 변수 정의
    int num;
    int name;

    // 멤버 함수 정의
    void Push(int value);
    int Pop();
};
```

```
class D_Queue
{
    // 멤버 변수 정의
    double num;
    double name;

    // 멤버 함수 정의
    void Push(double value);
    double Pop();
};
```

```
template <class T>// class 대신 typename도 가능
class Queue
{
    // 멤버 변수 정의
    T num;
    T name;

    // 멤버 함수 정의
    void Push(T value);
    T Pop();
};
```

클래스 템플릿으로 정의한 Stack 클래스

❖ 템플릿의 기본형(default) 지정 가능

❖ 클래스 템플릿은 <int>, <float>... 등과 같은 인수 생략 불가

```
template <class T=int> // T의 기본값은 int
class Stack {
protected:
    int m_size;
    int m_top;
    T *m_buffer;

public:
    Stack(int size = sz);
   ~Stack();
    void Push(T value);
    T Pop();
    ...
};

template <class T> // 함수 정의에도 작성 필요
void Stack<T>::Push(T value) {
                ...
}

template <class T>
T Stack<T> ::Pop() {
                ...
}
```

```
int main(){
   // int 타입을 다루는 스택 객체 생성
   Stack<int> iStack; // 함수 정의에 명시적 인수 지정
   // double 타입을 다루는 스택 객체 생성
   Stack<double> dStack;

   iStack.Push(3);
   int n = iStack.Pop();

   dStack.Push(3.5);
   double d = dStack.Pop();
}
```

클래스 템플릿 사용 예

❖ 개발자가 만든 코드- case: 2 type, 1 template parameter(TP)

```
template < class A, class B, int MAX > // 2 type, 1 TP
class TwoArray {
        // 중간 생략
        A arr1[ MAX ];
        B arr2[ MAX ];
};

int main(){
        TwoArray< char, double, 20 > arr;
}
```

❖ 컴파일러에 의해 작성된 클래스

```
class TwoArray_char_double_20 { // 임의로 만들어진 이름
        // 중간 생략
        char arr1[ 20 ];
        double arr2[ 20 ];
};
```

클래스 템플릿의 인스턴스화(구체화)

❖ 항상 명시적으로 지정

- 객체 생성 O, 템플릿의 파라미터 지정

```
Stack<int>  s1(10);              // T는 int 형
Stack<string> s2(5);             // T는 string 형
```

- 객체 생성 X, **포인터 혹은 레퍼런스를** 정의 및 템플릿의 파라미터 지정

```
Stack<char> *pStack = NULL; // T는 char 형
```

- typedef 문으로 클래스 템플릿의 별명을 정의하면서 템플릿의 파라미터 지정
  - typedef: type definition의 줄임말, 자료형의 새로운 이름(별명) 정의
  - 앞으로 "타입 Stack<Point>를 PointStack 타입으로 부르겠다", typedef 타입명 별명;

```
typedef Stack<Point>  PointStack;   // T는 Point 형
PointStack s4(20);                  // Stack<Point> 클래스의 객체 생성
```

클래스 템플릿의 사용

❖무엇이든지 타입으로써 사용 가능

```
void f1(Stack<int>& s);       // 함수의 인자로 사용

Stack<double>* f2();        // 함수의 리턴형으로 사용

class X {
protected:
    Stack<string> m_strStack;      // 멤버 객체의 데이터 형으로 사용
    ...
};

class MyStack : public Stack<int> {  ..  };    // 다른 클래스의 기본 클래스로 사용

Template <typename T>
class Array {
  protected:
     T arr[3];
    ..
};
Array< Stack<int> >  arra;                  // 다른 클래스 템플릿의 파라미터로 사용
```

템플릿의 특징

❖템플릿의 인스턴스화 이전에는 코드를 생성하지 않음

- 템플릿은 일종의 **설계도**
- 설계도의 완성은 **템플릿의 인스턴스화(구체화)**
- **컴파일러는** 구체화된 설계도에 따라 코드를 생성
- 설계도에 맞는 코드를 생성하기 때문에 컴파일 시간 증가(문제가 될 정도는 아님)

❖코드 크기를 최소화 함

- 작성된 인스턴스만 구현

❖프로그램이 *.cpp와 *.h 로 분할되어 작성된 경우, 템플릿은 *.h에 위치해야 함

- 다른 소스파일(*.cpp)에서 템플릿을 사용할 때 문제가 될 수 있음
- 구현코드도 *.h에 있어야함

목차

❖ 클래스 템플릿

❖ **STL**

STL(Standard Template Library)

❖STL = Standard Template Library

- 표준 템플릿 라이브러리
  - C++ 표준 라이브러리 중 하나
  - 대부분의 C++ 컴파일러는 STL을 지원
- 성능이 우수하고 안전성이 검증된 라이브러리
- 많은 템플릿 클래스(제네릭 클래스)와 템플릿 함수(제네릭 함수) 포함
  - 개발자는 이들을 이용하여 쉽게 응용 프로그램 작성
  - 기간도 단축하고 최소한의 노력만으로 원하는 기능을 구현 가능

❖STL의 구성: container, iterator, algorithm

STL의 구성(1)

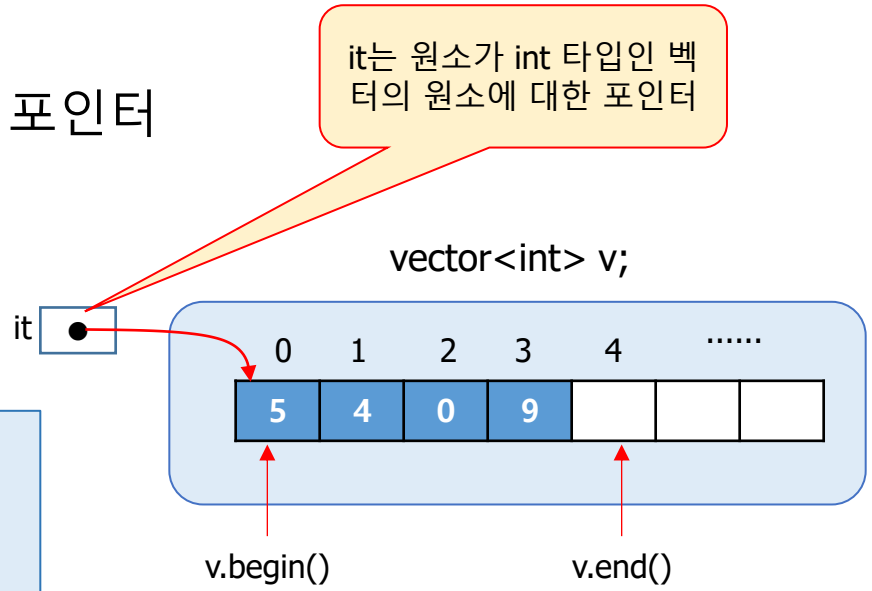
- **컨테이너** – 템플릿 클래스
  - 데이터를 담아두는 자료 구조를 표현한 클래스
  - 사용하고자 하는 클래스의 헤더파일 포함 필요
    - vector 클래스를 사용하려면 **#include <vector>**
  - 클래스의 이름은 표준 이름 공간(std)에 할당되어 있음
    - using namespace std;

컨테이너 클래스	설명	헤더 파일
vector	동적 크기의 배열을 일반화한 클래스	<vector>
deque	앞뒤 모두 입력 가능한 큐 클래스	<deque>
list	빠른 삽입/삭제 가능한 리스트 클래스	<list>
set	정렬된 순서로 값을 저장하는 집합 클래스, 값은 유일	<set>
map	(key, value) 쌍으로 값을 저장하는 맵 클래스	<map>
stack	스택을 일반화한 클래스	<stack>
queue	큐를 일반화한 클래스	<queue>

STL의 구성(2)

❖iterator

- 반복자라고도 부름
- 컨테이너의 각 요소들을 빠짐없이 순회 접근하기 위한 객체 포인터



❖iterator 변수 선언

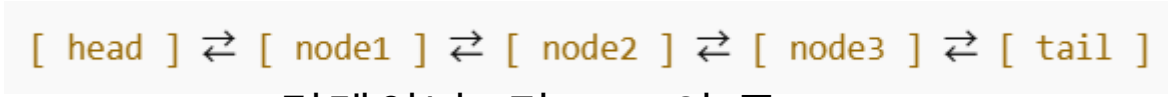
- 구체적인 컨테이너를 지정하여 반복자 변수 생성

```
vector<int> v = {5,4,0,9};
vector<int>::iterator it; //int 타입의 벡터를 가리키는 포인터
it = v.begin();
```

- begin() 과 end() 함수
  - STL 컨테이너가 공통으로 제공하는 함수
  - begin() 함수 : 첫 번째 원소를 가리키는 iterator를 리턴
  - end() 함수 : 마지막 원소를 가리키는 iterator를 리턴

STL의 list 사용하기

```
#include <list> // 1)
#include <iostream>

int main()
{
	// int 타입을 담을 이중 연결 리스트 생성
	std::list<int> intList; // 2)

	// 1 ~ 10까지 링크드 리스트에 넣는다.
	for (int i = 0; i < 10; ++i)
		intList.push_back(i); // 3)

	// 5를 찾아서 제거한다.
	intList.remove(5); // 3)

	// 링크드 리스트의 내용을 출력한다.
	std::list<int>::iterator it; // 4)

	for (it = intList.begin(); it != intList.end(); ++it){ // 5)
		std::cout << *it << "□n";
	}
	return 0;
}
```



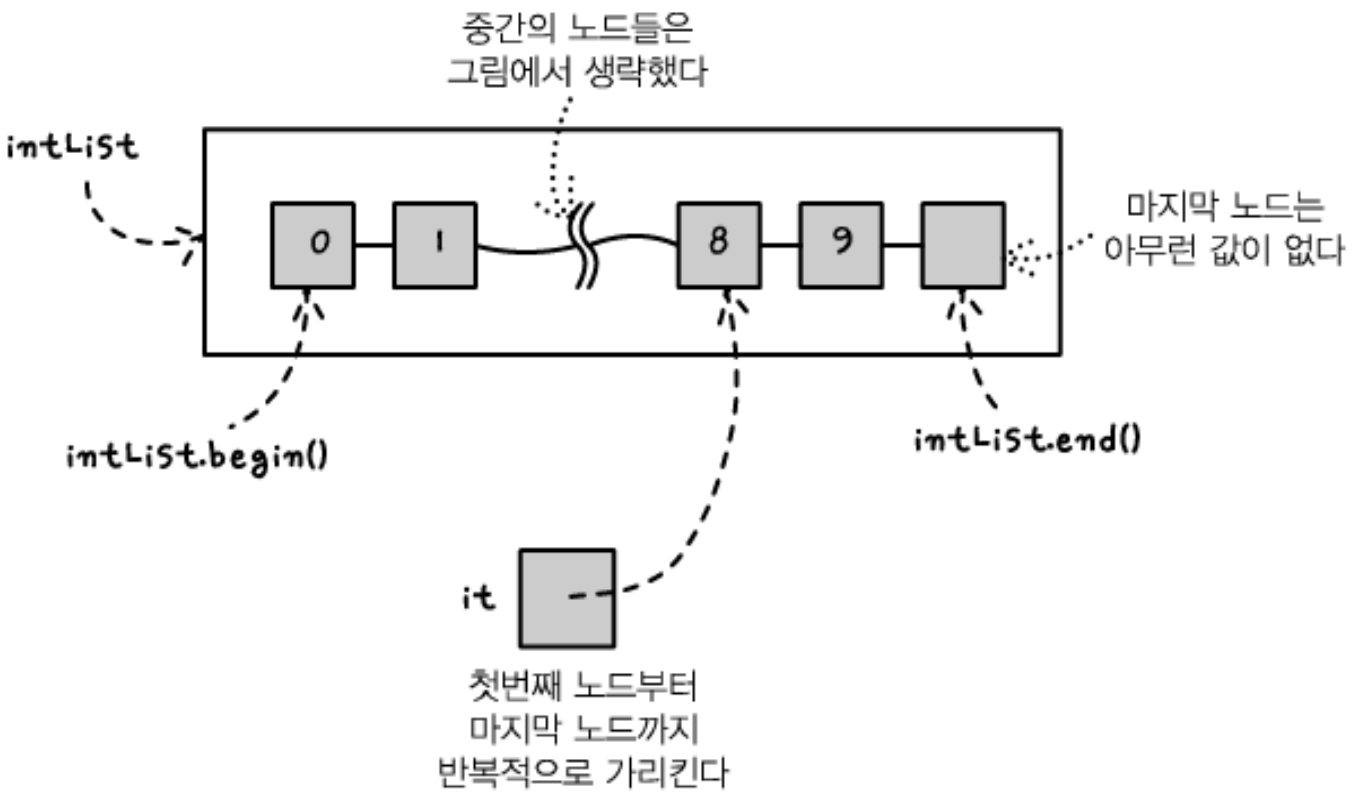
컨테이너 "리스트"의 구조

1) 헤더파일 추가
2) "list" 컨테이너 생성
3) 컨테이너에 값 저장 및 삭제
4) 리스트용 "iterator" 선언
5) iterator를 이용한 탐색

iterator 를 이용한 탐색

```
list<int>::iterator it;      // iterator 클래스 객체 생성
for (it = intList.begin(); it != intList.end();  ++it)
        cout << *it << "□n";
```

```
0
1
2
3
4
5
6
7
8
9
```



STL의 map 컨테이너

❖ **map 컨테이너:** key에 해당하는 value를 저장하고 찾을 수 있는 구조

❖ map<key, value>: 딕셔너리형 자료 구조

- ('키', '값')의 쌍을 원소로 저장하는 제네릭 컨테이너 : 동일한 '키'를 가진 원소가 중복 저장되면 오류 발생 ('키'로 '값' 검색)
- 자동 정렬된 상태에서 데이터를 저장하고, 키 값을 기준으로 빠르게, 검색 삭제 가능

멤버와 연산자 함수	설명
insert(pair<> &element)	맵에 '키'와 '값'으로 구성된 pair 객체 element 삽입
at(key_type& key)	맵에서 '키' 값에 해당하는 '값' 리턴
begin()	맵의 첫 번째 원소에 대한 참조 리턴
end()	맵의 끝(마지막 원소 다음)을 가리키는 참조 리턴
empty()	맵이 비어 있으면 true 리턴
find(key_type& key)	맵에서 '키' 값에 해당하는 원소를 가리키는 iterator 리턴
erase(iterator it)	맵에서 it가 가리키는 원소 삭제
size()	맵에 들어 있는 원소의 개수 리턴
operator[key_type& key]()	맵에서 '키' 값에 해당하는 원소를 찾아 '값' 리턴
operator=()	맵 치환(복사)

STL의 map 사용하기

1) 헤더파일 추가
2) "map" 컨테이너 생성
3) 영어, 한글 단어 쌍 저장
   - insert(n): STL 컨테이너에 새로운 원소를 추가하는 함수
   - **make_pair(a, b)**: a, b 값을 묶어, 한 개의 객체 생성
4) key가 "love"고 value가 "사랑"인 값 추가
5) value "사랑" 을 kor에 저장
   - key "love"가 존재하지 않으면, 생성 후 빈 문자열 저장
6) value "사랑" 을 kor에 저장
   - key "love"가 존재하지 않으면, **에러발생**

```
 #include <map> // 1)
sing namespace std;

map<string, string> dic;   // 2)

// 원소 저장
dic.insert(make_pair("love", "사랑")); // 3)
dic["love"] = "사랑";  // 4)

// 키 값에 해당하는 원소 리턴
string kor = dic["love"];     // kor은 "사랑"// 5)
string kor = dic.at("love"); // kor은 "사랑"// 6)
```

make_pair() 함수

❖ make_pair() : 템플릿 함수

- map.h 혹은 utility.h 헤더파일을 포함시켜야 사용가능
- 두 개의 값을 받아 pair 객체를 생성하는 함수
- 주로 map과 같이 키-값 쌍을 저장하는 컨테이너에 데이터를 추가할 때 사용

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(T1 t1, T2 t2) {...};
```

make_pair() 함수의 형태

```
pair<int, string> p = make_pair(1,"hello");
cout << p.first << ": " << p.second << endl;
```

make_pair() 함수의 활용

pair 클래스

❖ **std::pair : 템플릿 클래스**

- 2개의 서로 다른 타입의 값을 하나로 묶을 수 있는 자료구조
- 각각의 값은 first와 second라는 이름으로 접근.

```
namespace std {
    template <typename T1, typename T2>
    struct pair {
        T1 first;  // 첫 번째 값 (T1 타입)
        T2 second; // 두 번째 값 (T2 타입)

        // 생성자, 연산자 오버로드 등 다양한 기능을 제공
    };
}
```

```
std::pair<int, std::string> p(1, "Hello");
std::cout << p.first << " " << p.second;
```

STL의 map 사용하기 해석

3) 컨테이터에 영어, 한글 쌍 저장

- make_pair(): 2개의 인수로 pair 객체 생성 및 반환
- insert(): pair 객체를 map 컨테이터에 저장

```
 #include <map> // 1)
sing namespace std;

map<string, string> dic;   // 2)

// 원소 저장
dic.insert(make_pair("love", "사랑")); // 3)
dic["love"] = "사랑";  // 4)

// 키 값에 해당하는 원소 리턴
string kor = dic["love"];      // kor은 "사랑"// 5)
string kor = dic.at("love");  // kor은 "사랑"// 6)
```

pair 클래스의 활용

❖ **const auto& pair**

- 범위 기반 for문을 사용할 때, std::map이나 다른 컨테이너의 원소를 순회하는 데 사용되는 표현식
- iterator 대신 보다 간단하고 직관적인 방식으로 std::map이나 다른 컨테이너의 원소를 순회할 수 있음.

```
map<string, int>::iterator it;
for (it = studentScores.begin(); it != studentScores.end(); ++it)
        std::cout << it→first << ": " << it→second << "\n";
```

=

```
for (const auto& pair : studentScores) {
        cout << pair.first << ": " << pair.second << "\n";
    }
```

STL 컨테이너의 장점

❖각 컨테이너가 제공하는 인터페이스는 동일

- push_back, back, begin, end 등의 멤버 함수는 다른 STL 컨테이너에서도 동일하게 제공

❖최소한의 코드 수정으로 컨테이너를 다른 것으로 바꿀 수 있음

- 프로그램의 개발이나 유지 보수 시간을 단축하는 데 큰 도움이 됨

STL의 구성(3)

❖ **알고리즘(Algorithm)**

❖STL이 제공하는 범용 함수

- 템플릿 함수
- 전역 함수
  - STL 컨테이너 클래스의 멤버 함수가 아님
- 주로 iterator와 함께 사용

copy	merge	random	rotate
equal	min	remove	search
find	move	replace	sort
max	partition	reverse	swap

알고리즘

❖ sort() 함수: 컨테이너의 요소를 정렬하기 위한 함수, 기본적으로 오름차순 정렬

- 두 개의 매개 변수
  - 첫 번째 매개 변수 : 정렬을 시작한 원소의 주소
  - 두 번째 매개 변수 : 정렬할 범위의 마지막 원소 다음 주소

```
vector<int> v;
...
sort(v.begin(), v.begin()+3); // v.begin()에서 v.begin()+2까지, 처음 3개 원소 정렬
sort(v.begin()+2, v.begin()+5); // 벡터의 3번째 원소에서 v.begin()+4까지, 3개 원소 정렬
sort(v.begin(), v.end()); // 벡터 전체 정렬
```

sort() 함수를 이용한 vector 정렬

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

void main() {
	// 동적 배열을 생성해서 임의의 영문자를 추가한다.
	vector<char> vec;
	vec.push_back( 'e');
	vec.push_back( 'b');
	vec.push_back( 'a');
	vec.push_back( 'd');
	vec.push_back( 'c');

	// sort() 함수를 사용해서 정렬한다.
	sort( vec.begin(), vec.end() );

	// 정렬 후 상태를 출력한다.
	cout << "vector 정렬 후□n";
	vector<char>::iterator it;
	for (it = vec.begin(); it != vec.end(); ++it)
		cout << *it;
```

```
	// 이번에는 배열을 정렬해보자
	// 임의 문자열을 넣은 배열을 만든다

	char arr[5] = {'d', 'c', 'b', 'a', 'e'};

	// sort() 함수를 사용해서 정렬한다.
	sort( arr, arr+5 );

	// 정렬 후 상태를 출력한다.
	cout << "배열 정렬 후□n";
	for (char* p = arr; p != arr + 5; ++p)
		cout << *p;
}
```

제너릭(generic) 프로그래밍

❖ 제너릭 프로그래밍
- 정보의 타입과 정보를 처리하는 알고리즘을 분리하여 서로 독립적으로 관리할 수 있는 프로그래밍 방식
  - 컨테이너 : '정보의 타입'
  - sort() 함수를 비롯한 알고리즘 함수 : '정보를 처리하는 알고리즘'

❖ 제너릭 프로그래밍 목표
- 정보의 타입과 관계없이 동일한 방법으로 처리할 수 있는 알고리즘을 구현하는 것

제너릭 프로그래밍의 장점

- 타입과 알고리즘간의 불필요한 연관성 제거(decoupling)
  - vector 클래스의 세부 구현을 변경했다고 해도 sort() 함수는 영향을 받지 않음
- 재사용성(reusability) 증가
- 확장의 용이성
  - 공통적인 인터페이스 : begin(), end() 등
  - 새로운 컨테이너 클래스를 만들 때 이 공통의 인터페이스를 유지한다면 STL의 모든 알고리즘을 사용할 수 있다.

끝?