C/C++ Day 4

윤광진, 광주과학기술원 02/08/2018

1 템플릿

매트랩이나 파이선에 익숙하다면 변수의 타입을 미리 지정할 필요 없이 사용자의 실행에 맞춰 자동으로 타입을 정하는 것을 보았을 것이다. C++에서도 비슷한 기능을 구현할 수 있는데 바로 템플릿이다. 이것을 이용하면 함수나 클래스를 변수타입에 따라 개별적으로 작성하지 않고도 여러 변수타입에서 동작 할 수 있다. 템플릿에는 함수 템플릿과 클래스 템플릿이 있다.

1.1 함수 템플릿

여러 변수 타입(**int, float, char ....**)들을 템플릿 인자로 받아 함수 내부에서 활용할 수 있게 한것이다. 템플릿 인자는 **<_>** 으로 받는다. 즉, 여러 변수타입에 대해 동작하는 함수를 템플릿으로 만들수 있다. 템플릿의 선언은 Listing 1과 같이 한다.

Listing 1: Template

```
1 template <typename id1, typename id2, ... > function_declaration;
```

id는 identifier(식별자)이다. 이번엔 예를 들어 두 변수를 입력받아 덧셈을 하는 함수를 템플릿으로 작성해보자.

Listing 2: Template

```
1 template <typename T>
2 T add(T a, T b)
3 {
4     return a + b;
5 }
```

Listing 2의 **add** 함수는 아직 정해지지 않은 자료형 **T**를 정의하고 **T** 타입의 변수 **a**와 **b**를 더해서 **T** 타입으로 반환하는 함수다. 그럼 **T**의 자료형은 언제 결정 지어지는 것일까? 그것은 **add** 함수가 사용되는 시점에 결정되어진다. 예를들어 **add** 함수를 사용해보자.

Listing 3: Template

```
1 #include <iostream>
2 using namespace std;
3 void main()
4 {
5     cout << add(1, 2) << endl;
6     cout << add(1.0, 2.0) << endl;
7     cout << add<int>(1, 2.0) << endl;
8     cout << add<double>(1, 2.0) << endl;
```

```
9       cout << (int)add('1', '2') << endl;   // ascii 49, 50
10  }
```

5번줄은 **add**함수를 사용할때 정수를 이용했으므로 **T**가 **int**로 변형된다. 6번줄은 **double** 형으로 사용된다. 7번줄은 프로그래머가 **add** 함수가 **int**로 컴파일 되도록 명시했으므로 입력이 **double** 이었어도 **int** 형으로 변환되어 사용된다. 8번줄은 **double**로 사용되도록 명시한것이다. 9번줄은 **add** 함수를 **char**형으로 사용하고 반환값을 **int**형으로 변환하여 출력하고있다. 이 **add** 함수가 받는 두 개의 인자는 항상 동일한 타입이어야한다. 두 인자 모두 같은 **T** 타입이기 때문이다. 따라서 **add(1, 2.0)** 같이 쓰려고 하면 오류가 난다. 7-8번줄이 오류가 나지 않은 이유는 **T**의 타입을 명시했기 때문이다.

만약 **add** 함수의 두 인자의 타입을 다르게 쓰고 싶다면 어떻게 어떻게 하면 될까? Listing 1의 함수 템플릿 정의 법을 보면 **typename**을 여러번 할수 있게 되어있다.

Listing 4: Template

```
1  template <typename T1, typename T2>
2  T1 add(T1 a, T2 b)
3  {
4      return a + b;
5  }
```

Listing 4의 **add** 함수는 **T1, T2** 두개의 템플릿을 함수의 파라미터로 받고 **T1** 타입으로 반환을 한다. **T1**과 **T2**의 자료형은 같아도 되고 아니어도 된다.

1.2 클래스 템플릿

클래스 템플릿은 말그대로 클래스에 템플릿을 적용하는 것이다. 클래스 템플릿도 함수 템플릿과 마찬가지 형태로 템플릿을 선언한다.

Listing 5: Template

```
1  template <typename T>
2  class Data
3  {
4      T data;
5  public:
6      Data(T d) { data = d; };
7      void setData(T d) { data = d; };
8      T getData() { return data; };
9  };
```

Listing 5에 템플릿을 사용한 **Data** 클래스를 정의하였다. 이 클래스는 별다른 기능은 없고 **T** 타입의 데이터 하나를 저장할 수 있는 클래스이다. 그럼 이 클래스를 사용해보자.

Listing 6: Template

```
1  void main()
```

```
2  {
3      Data<int> d1(1);
4      Data<char> d2('a');
5
6      cout << d1.getData() << endl;
7      cout << d2.getData() << endl;
8      d2.setData('2');
9      cout << d2.getData() << endl;
10 }
```

Listing 6의 3,4번 줄에 **Data** 클래스의 사용법이 나와있다. 함수 클래스와 다른 점은 사용할때 **T**의 타입명을 명시해줘야한다는 것이다.

템플렛을 처리해 주는 것은 컴파일러이기 때문에 템플렛 클래스는 선언과 정의를 다른 파일에서 할 수 없다. 즉, 하나의 헤더파일 안에 선언과 정의가 함께 있어야한다.

2 Standard Template Library

STL(Standard Template Libraray)은 C++ 클래스 템플렛들의 집합으로 자주 쓰이는 자료 구조(연결 리스트, 스택, 큐, 힙 등)와 알고리즘이 구현되어있다. STL에는 데이터를 저장하고 관리할 수 있는 유용한 컨테이너들이 있는다. 여기서는 컨테이너들 중 **array**와 **vector**를 살펴보자. 그리고 STL의 알고리즘 사용법도 공부해 보겠다.

2.1 array

array[1]는 고정 크기의 선형 컨테이너이다. 선형 컨테이너란 말은 데이터가 메모리에 연속적으로 배치된다는 의미이다. **array**는 고정 크기를 가지기 때문에 동적으로 크기를 줄이거나 늘릴 수 없다.

array에서 지원하는 함수 및 기능들을 알아보자.

- **at()** : **array** 원소에 접근 한다.
- **operator[]** : **array** 원소에 접근 한다.
- **front()** : **array**의 첫번째 원소를 반환한다.
- **back()** : **array**의 마지막 원소를 반환한다.
- **size()** : **array**의 원소 개수를 반환한다.
- **swap()** : **array**의 내용을 다른 **array**와 맞바꾼다. 맞바꾸려는 **array**는 자료형과 크기가 같아야한다.

다음 Listing 7을 실행해 보자.

[1] **array** 설명에는 약간의 C++11 스타일이 포함될 수 있음

Listing 7: **array**

```
1  #include <iostream>
2  #include <array>
3  using namespace std;
4  void main()
5  {
6      array<int, 3> arr1;
7      arr1.at(0) = 0;
8      arr1[1] = 1;
9      arr1[2] = 2;
10
11     const int n = 3;
12     array<int, n> arr2;
13
14     for (int i = 0; i < arr2.size(); ++i)
15         arr2.at(i) = i + 3;
16
17     for (int i = 0; i < arr1.size(); ++i)
18         cout << arr1.at(i) << endl;
19     cout << endl;
20     for (int i = 0; i < arr2.size(); ++i)
21         cout << arr2.at(i) << endl;
22     cout << endl;
23
24     cout << arr1.front() << ", " << arr1.back() << endl;
25     arr1.swap(arr2);
26
27     for (int i = 0; i < arr1.size(); ++i)
28         cout << arr1.at(i) << endl;
29     cout << endl;
30     for (int i = 0; i < arr2.size(); ++i)
31         cout << arr2.at(i) << endl;
32     cout << endl;
33 }
```

2.2 vector

vector는 STL에서 자주쓰이는 컨테이너다. **vector**는 자동으로 크기를 늘리거나 줄일수 있어 데이터를 삽입하거나 삭제 할 수 있다. **vector**에서 지원하는 함수 및 기능을 보자.

- **begin()** : **vector**의 첫번째 원소를 가리키는iterator를 반환한다.
- **end()** : **vector**의 마지막 원소를 가리키는iterator를 반환한다.

- **rbegin()** : **vector**의 마지막 원소를 가리키는 역방향 iterator를 반환한다. 뒤에서 앞으로 이동한다.
- **rend()** : **vector**의 첫번째 원소를 가리키는 역방향 iterator를 반환한다. 앞에서 뒤로 이동한다.
- **size()** : **vector** 원소의 갯수를 반환한다.
- **max_size()** : **vector**가 가질수 있는 최대 원소 개수를 반환한다.
- **operator[]** : **vector**의 원소에 접근한다.
- **at()** : **vector**의 원소에 접근한다.
- **push_back()** : **vector**의 끝에 원소를 더한다.
- **pop_back()** : **vector**의 마지막 원소를 제거 한다.
- **insert()** : **vector**에 원소를 더한다.
- **erase()** : **vector**의 원소를 제거한다.
- **swap()** : **vector**를 다른 **vector**와 맞바꾼다.
- **clear()** : **vector**의 모든 데이터를 지운다.

Listing 8에 **vector** 사용법 예제가 있다.

Listing 8: **vector**

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  void main()
5  {
6      int n = 5;
7      vector<int> v1;
8      for (int i = 0; i < n; ++i)
9          v1.push_back(i);
10
11     cout << "max_sz: " << v1.max_size() << endl;
12     vector<int>::iterator it;
13     cout << "v1 foward: ";
14     for (it = v1.begin(); it != v1.end(); ++it)
15         cout << *it << " ";
16     cout << endl;
17
18     vector<int>::reverse_iterator rit;
19     cout << "v1 reverse: ";
```

```
20      for (rit = v1.rbegin(); rit != v1.rend(); ++rit)
21          cout << *rit << " ";
22      cout << endl;
23
24      for (int i = 0 ; i < v1.size(); ++i)
25          v1.at(i) = i * 10; //v1[i] = i * 10;
26
27      cout << "v1*10 : ";
28      for (int i = 0; i < v1.size(); ++i)
29          cout << v1[i] << " "; // cout << v1.at(i) << " ";
30      cout << endl;
31
32      v1.pop_back();
33      cout << "v1 pop_backed: ";
34      for (int i = 0; i < v1.size(); ++i)
35          cout << v1.at(i) << " ";
36      cout << endl;
37
38      vector<int> v2(3, 100);
39      cout << "v2 : ";
40      for (int i = 0; i < v2.size(); ++i)
41          cout << v2.at(i) << " ";
42      cout << endl;
43      v1.swap(v2);
44
45      cout << "v1 : ";
46      for (int i = 0; i < v1.size(); ++i)
47          cout << v1.at(i) << " ";
48      cout << endl;
49      cout << "v2 : ";
50      for (int i = 0; i < v2.size(); ++i)
51          cout << v2.at(i) << " ";
52      cout << endl;
53  }
```

2.2.1 **insert, erase**

여기서는 **vector**의 **insert**와 **erase**의 사용법을 알아본다. 우선 **insert** 함수의 사용법은 Listing 9와 같다.

Listing 9: **vector**

```
1  iterator insert (iterator position, const value_type& val);
```

```
2 void insert (iterator position, size_type n, const value_type& val);
3 void insert (iterator position, InputIterator first, InputIterator last);
```

insert는 **position** 위치 앞에 주어진 원소를 삽입한다. **position**은 iterator이다. Listing 9의 첫번째 줄은 **position** 앞에 **val**을 삽입하고 두번째 줄은 **position** 앞에 **val**을 **n**번 삽입을 한다. 세번째 줄은 **position** 앞에 입력의 **first**부터 **last**까지를 삽입한다. Listing 10를 보자.

Listing 10: **vector**

```
1 #include <iostream>
2 #include <vector>
3
4 int main ()
5 {
6     vector<int> myvector(3, 100);
7     vector<int>::iterator iter;
8
9     iter = myvector.begin();
10    iter = myvector.insert(iter, 200);
11
12    myvector.insert(iter, 1, 300);
13
14    iter = myvector.begin();
15
16    vector<int> anothervector(2, 400);
17    myvector.insert(iter + 2, anothervector.begin(), anothervector.end());
18
19    int myarray[] = { 501,502,503 };
20    myvector.insert(myvector.begin(), myarray, myarray + 3);
21
22    cout << "myvector contains:";
23    for (iter = myvector.begin(); iter<myvector.end(); iter++)
24        cout << ' ' << *iter;
25    cout << '\n';
26 }
```

erase는 **vector**의 원소를 삭제하는 함수다. Listing **??**에 함수 사용법을 나타냈다.

Listing 11: **vector**

```
1 iterator erase (iterator position);
2 iterator erase (iterator first, iterator last);
```

Listing 11의 첫째줄은 **position** 위치의 원소를 지운다. 둘째줄은 **first**부터 **last**까지 원소를 지운다. Listing 12 예제를 보자.

Listing 12: **vector**

```
#include <iostream>
#include <vector>

int main ()
{
    vector<int> myvector;

    for (int i = 1; i <= 10; i++)
        myvector.push_back(i);

    // erase the 6th element
    myvector.erase(myvector.begin() + 5);

    // erase the first 3 elements:
    myvector.erase(myvector.begin(), myvector.begin() + 3);

    std::cout << "myvector contains:";
    for (unsigned i = 0; i<myvector.size(); ++i)
        std::cout << ' ' << myvector[i];
    std::cout << '\n';
}
```

2.3 Algorithm

STL에서는 유용한 알고리즘을 제공한다. 여기서는 그 중에서 **sort**(정렬), **max(min)_element**, **find**, **count**, **binary_search**에 대해서 알아본다.

Listing 13: **algorithm**

```
sort(first_iterator, last_iterator)
max_element (first_iterator, last_iterator)
min_element (first_iterator, last_iterator)
find(first_iterator, last_iterator, x)
count(first_iterator, last_iterator,x)
binary_search(first_iterator, last_iterator, x)
```

Listing 13의 첫번째 줄은 정렬 알고리즘이다. 정렬하고자 하는 컨테이너의 시작 위치와 끝 위치를 넘겨주면 된다. **sort**는 디폴트로 함수로 넘겨진 컨터이너를 오름차순으로 정렬한다[2]. 2번, 3번 줄은 최대 및 최소값을 찾는 함수이다. 이들 함수의 반환은 iterator이기 때문에 최대 최소 값을 받으려면 포인터 연산자(*)를 사용해야함을 주의하자. 4번 줄의 **find** 함수는 찾고자하는 원소와

[2]내림차순으로 정렬하는 법도 예제 Listing 14에 포함시키겠다.

같은것이 컨테이너에에 있는지 확인하는 함수다. 원소를 찾았다면 컨테이너의 위치를 반환하고 못 찾았다면 컨테이너의 제일 끝 위치를 반환한다[3]. 5번줄의 **count**는 찾고자 하는 원소가 컨테이너에 몇개 있는지를 세는 함수다. 6번줄은 **binary_search** 함수로 정렬되어진 컨테이너를 입력으로 받아서 찾고자하는 원소를 탐색한다. 알다시피 이진 탐색 알고리즘이므로 정렬이 완료된 컨테이너를 입력으로 주어야한다. 찾고자 하는 원소가 있으면 **true**를 아니면 **false**를 반환한다.

Listing 14: **algorithm**

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <random>
5  using namespace std;
6
7  bool mycomp(int l, int r)
8  {
9      return l > r;
10 }
11
12 void main()
13 {
14     int n = 10;
15     default_random_engine gen;
16     uniform_int_distribution<> dis(1, 100);
17     vector<int> vec;
18     for (int i = 0; i < n; ++i)
19         vec.push_back(dis(gen));
20
21     cout << "vector: ";
22     for (int i = 0; i < n; ++i)
23         cout << vec[i] <<" ";
24     cout << endl;
25
26     // max, min elements
27     cout << "Maximum element of vector is: ";
28     cout << *max_element(vec.begin(), vec.end()) << endl;
29     cout << "Minimum element of vector is: ";
30     cout << *min_element(vec.begin(), vec.end()) << endl;
31
32     // sort
33     sort(vec.begin(), vec.end());
34     cout << "vector: ";
```

[3]컨테이너의 끝위치(last)에는 데이터가 없다.

```
35      for (int i = 0; i < n; ++i)
36          cout << vec[i] << " ";
37      cout << endl;
38
39      // find
40      int q = 101;
41      vector<int>::iterator iter;
42      iter = find(vec.begin(), vec.end(), q);
43      if (iter == vec.end())
44          cout << "not found " << q << endl;
45      else
46          cout << "found " << q << endl;
47
48      // count
49      q = vec[0];
50      vec.push_back(q);
51      cout << q << " occured: " <<count(vec.begin(), vec.end(), q) << endl;
52
53      // sort descending order
54      sort(vec.begin(), vec.end(), mycomp);
55      cout << "vector: ";
56      for (int i = 0; i < n; ++i)
57          cout << vec[i] << " ";
58      cout << endl;
59
60      // sort ascending order again
61      sort(vec.begin(), vec.end());
62
63      // binary search
64      q = vec[0];
65      if(binary_search(vec.begin(), vec.end(), q))
66          cout << "found " << q << endl;
67      else
68          cout << "not found " << q << endl;
69  }
```