Standard Template Library

HCI Programming 2 (321190)
2007년 가을학기
9/17/2007
박경신

Overview

- Template
- Function Template
- Explicit Specification
- Class Template
- Standard Template Library (STL)
  - Containers
  - Algorithms
  - Iterators
  - Functors

Template

- C++ Standard Template Library (STL) 클래스는 템플릿 기반의 클래스
- 템플릿 (Template)
  - 템플릿이란 다른 데이터 타입에 적용 가능한 함수와 클래스를 만들어낼 수 있는 틀
  - 예를 들어, int, long, 또는 double 데이터 타입으로 사용할 수 있는 linked list 클래스를 생성하는 것
- 템플릿의 종류
  - Function Template
  - Class Template

Function Template

- int 와 long 타입을 지원하는 Swap함수가 두 개 필요하다면? 오버로딩(overloading)된 함수를 제공

```
void Swap(int &i, int &j)
{
   int  t = i;
   i = j;
   j = t;
}
void Swap(long &i, long &j)
{
   long  t = i;
   i = j;
   j = t;
}
```

Function Template

- 단순히 두 종류의 인자를 전달하기 위해 두 가지 함수를 만들어야 하는가?
- 템플릿을 이용하면 이런 문제 해결 가능
- Swap이라는 함수를 만들어낼 수 있는 틀 (즉 템플릿)을 만들면 변수의 타입에 따라 적절한 함수를 만들어냄

```
void Swap(SwapType &i, SwapType &j) {
   SwapType t = i;
   i = j;
   j = t;
}
```


Function Template

- 템플릿화 시키는 방법

```
template <class SwapType>
void Swap(SwapType &i, SwapType &j) {
   SwapType t = i;
   i = j;
   j = t;
}
template <typename SwapType>
void Swap(SwapType &i, SwapType &j) {
   SwapType t = i;
   i = j;
   j = t;
}
```


Function Template

- 전 슬라이드는 함수가 아니라 템플릿에 불과함
- 실제 함수는 템플릿으로부터 컴파일러에 의해 자동으로 만들어짐

```
#include <stdio.h>

template <typename SwapType>
void Swap(SwapType &i, SwapType &j) {
   SwapType t = i;
   i = j;
   j = t;
} // Swap
void main() {
   int i = 2, j = 3;
   Swap(i, j);
   printf("i = %d, j = %d", i, j);
}
```


Function Template

- 아래의 경우 컴파일러가 char, int, double의 세 개의 Swap 함수를 생성할 것임

```
void main() {
      char a = 'a';
      char b = 'b';
      int i = 2, j = 3;
      double c = 1.2, d = 3.4;

      Swap(a, b);
      Swap(i, j);
      Swap(c, d);
}
```

Function Template

- 템플릿은 형의 리스트 (type list)를 지정할 수 있음

```
template <class T, class U, class V>
void Func (T t, U u, V v) {
   …
}
```


Function Template

```
#include <iostream>

template <typename T>
T median (T x, T y, T z) {
   if (x < y)
      return (y < z ? y : x < z ? z : x);
   else
      return (x < z ? x : y < z ? z : y);
}

int main() {
   cout << median(10, 4, 5) << endl;
   cout << median(15L, 4L, 3L) << endl;
   cout << median('r', 'g', 'h') << endl;
   cout << median(1.4, 1.2, 0.9) << endl;
   cout << median(1.1f, 1.3f, 1.4f) << endl;
   cout << median(1, 1.2, 'a') << endl; // 컴파일 에러 – Type이 다름
}
```


Function Template

- 템플릿의 파라미터로 상수 (value)를 지정할 수 있음

```
#include <iostream>
#include <string>

template <typename T, typename U, typename V>
void Func (T t, U u) {
   T buf[V];
   …
}
void main () {
   Func<int, float, 100>(30, 50.0f);
   Func<int, int, 50>(40, 60);
   …
}
```


Function Template

```
#include <iostream>

template <typename T1, typename T2, typename T3>
T1 median (T1 x, T2 y, T3 z) {
   if (x < y)
      return (y < z ? y : x < z ? z : x);
   else
      return (x < z ? x : y < z ? z : y);
}

int main() {
   cout << median(10, 4, 5) << endl;
   cout << median(15L, 4L, 3L) << endl;
   cout << median('r', 'g', 'h') << endl;
   cout << median(1.4, 1.2, 0.9) << endl;
   cout << median(1.1f, 1.3f, 1.4f) << endl;
   cout << median(1, 1.2, 'a') << endl;
}
```

Explicit Specification

- 여태까지 본 예제는 인자를 통해 템플릿 매개 변수의 타입을 유추하는 것이 가능한 경우였음
- 때에 따라서는 템플릿 매개 변수를 파악하기 어렵거나 추론된 것이 원하는 타입이 아닐 경우도 있음

```
template <typename rst_t>
rst_t f() {
   …
}

template <typename T>
void g() {
   T a;
   …
}
```

```
template <typename rst_t, typename lt>
rst_t h(lt l) {
      ….
}

int i = f(); // rst_t를 유추하기 어려움
g();  // T를 유추하기 어려움
char c = h(10);  // rst_t를 유추하기 어려움
```


Explicit Specification

```
#include <iostream>
template <typename T>
T cast(int s) {
    return (T)s;
}
template <typename T>
void func(void) {
    T v;
    cin >> v;
    cout << v;
}
void main() {
    unsigned i = cast<unsigned> (1234);  // cast(1234)로는 T 타입을 모름
    double d = cast<double> (5678);      // 따라서 명시적으로 호출해야 함
    cout << "i=" << i << " d=" << d << endl;
    func<int> ();                        // func()로는 T 타입을 모름
}
```


Explicit Specification

- 이런 경우에는 템플릿의 인자를 직접 지정 가능 (명시적 템플릿 인자 – Explicit template arguments)
- 명시적 기입 방법은 템플릿을 사용할 때 템플릿 함수의 이름 바로 뒤에 템플릿이 선언될 때와 마찬가지의 순서로 기호 <를 열고 안에 쉼표를 이용해서 명시할 타입을 기입

```
#include <iostream>
template <typename rst_t, typename lt, typename rt>
rst_t plus(lt l, rt r) {
   rst_t rst = l + r;
   return rst;
}
int main() {
   cout << plus<int, int, int> (1, 3) << endl;       // 즉, int plus(int, int)
   cout << plus<int, int, int> (1.1, 3.5) << endl;
   cout << plus<double, int, double> (1.1, 3.5) << endl;
}
```


Class Template

- 타입만 다른 클래스는 템플렛을 이용함

```
class CStack {
  public:
    CStack(int s) {
       data = new int[size = s];
       sp = 0;
    }
    ~Cstack() {
       delete[] data;
    }
    void Push(int d);
    int Pop();
  private:
    int *data;
    int size, sp;
};
```

Class Template

```
template <class T>
class CStack {
   public:
      CStack(int s) {
         data = new T[size = s];
         sp = 0;
      }
      ~Cstack() {
         delete[] data;
      }
      void Push(T d);
      T Pop();
   private:
      T *data;
      int size, sp;
};
```


Class Template

- 외부에서 멤버를 따로 정의하고자 할 때는 멤버 함수가 템플릿이라는 것을 알려주어야 하는 과정이 추가로 필요

```
template <class T>
void CStack<T>::Push(T d) {
   …
}
template<class T>
T CStack<T>::Pop() {
   …
}
```

- 클래스가 사용하는 형은 함수와는 달리 컴파일 시간에 알려지지 않음
- 원하는 형의 스택을 만들려면 프로그래머가 명시적으로 형을 적어야만 함

```
CStack<int> s(100);
```


Class Template

```
template <typename T>
class X {
      public:
         X(T d ) : data(d) { }
         void f () {
            cout << typeid(data).name() << " : " << data << endl;
         }
      private:
         T data;
};
void main () {
    X<int> x(10);
    x.f();
    X<char> y('a');
    y.f();
}
```


Class Template

```
template <class T, class TypeSP>
class CStack {
   public:
      CStack(int s) { data = new T[size = s]; sp = 0; }
      ~CStack() { delete [] data; }
      void Push(T d);
      T Pop();
   private:
      T *data;
      TypeSP size, sp;
};
```

Class Template

```
template <class T, class TypeSP>
void CStack<T, TypeSP>::Push(T d) {
    …
}

template<class T, class TypeSP>
CStack<T, TypeSP>::Pop() {
    …
}

int main() {
    CStack<char, int> s(10);
    CStack<int, int> t(10);
    …
}
```


Standard Template Library (STL)

- Standard Template Library (STL) - 일반적 프로그래밍을 목적으로 템플릿을 기반으로 한 표준 라이브러리의 일부
- 컨테이너, 알고리즘, 반복자, 함수자로 구성됨
- 컨테이너 (Containers)
  - 스택(Stack), 연결 리스트(Linked List), 큐(Queue), 균등 트리(Balanced Tree) 등의 데이터 관리 구조
- 알고리즘 (Algorithms)
  - 원소 전체 혹은 일부 구간에 적용하여 복사, 전환, 병합, 정렬 등의 연산
- 반복자 (Iterators)
  - 포인터의 개념을 한층 일반화시킨 지능화된 포인터
- 함수자 (Functor) 혹은 함수 객체 (Function Object)로 불림
  - operator()를 오버로딩한 클래스

Header and Namespace

- 표준 헤더(header)의 중요한 두 가지
  - 확장자가 없음
  - 헤더가 파일이 아닐 수도 있음
- 표준 라이브러리의 네임 스페이스 (Namespace)
  - std
  - using namespace std; 라는 선언의 의미는 표준 라이브러리의 네임스페이스를 열겠다는 것

Containers

- 시퀀스 컨테이너 (Sequence Containers)
  - 동일한 타입의 객체들의 집합을 선형의 배열로 구성
  - vector, list, deque
- 조합 컨테이너 (Associative Containers)
  - 일정한 규칙에 따라 자료를 조직화하여 관리
  - set, multiset, map, multimap
- 컨테이너 어댑터 (Container Adaptors)
  - 시퀀스 컨테이너를 변형하여 자료를 미리 정해진 일정한 방식에 따라 관리
  - stack, queue, priority queue
  - 반복자 (Iterator)를 지원하지 않음

Algorithms

- STL은 일반화된 알고리즘 함수를 제공하고 있음
- 변형 알고리즘
  - 입력된 반복자가 가리키는 곳에 변화를 주는 변형 알고리즘
  - random_suffle, reverse, rotate, transform, replace, fill, generate, unique, remove, copy, merge, swap, 등
- 불변형 알고리즘
  - 반복자가 가리키는 곳에 변화를 주지 않는 불변형 알고리즘
  - find, search, for_each, mismatch, equal, count, 등
- 정렬 알고리즘
  - 정렬에 관련된 알고리즘
  - sort, stable_sort, nth_element, lower_bound, upper_bound, 등

Iterators

- 포인터의 개념을 한층 일반화시킨 지능화된 포인터
- 내부적으로는 실제 포인터를 관리하는 클래스이며, 포인터를 이용해 다양한 형태의 데이터 구조를 일관된 방법으로 원소에 접근할 수 있는 방법을 제공
- 포인터를 사용하는 데 쓰는 연산자인 *, ->, ++, -- 등을 반복자에 맞게 오버로딩하였기 때문에 일반 포인터와 같은 방식으로 사용 가능
- 연산
  - itr++; 다음 위치로 움직일 수 있도록 하는 연산자
  - itr--; 이전 위치로 움직일 수 있도록 하는 연산자
  - *itr; 현 itr 위치에 저장된 원소의 레퍼런스를 반환
  - itr1 == itr2; itr1과 itr2이 같은 위치를 가리키면 true 반환
  - itr1 != itr2; itr1과 itr2이 다른 위치를 가리키면 true 반환

Functor (Function Object)

- 함수자 (Functor, Function object)는 operator()가 구현된 클래스 객체
- 알고리즘에 보낼 단 하나의 연산 함수만 가지고 있음
- 함수의 역할을 하면서 상태를 가질 수 있음
- 미리 정의된 함수자 - <functional> 헤더
  - 산술 연산자
    - time, divides, modulus, 등
  - 비교 연산자
    - greater, less, less_equal, 등
  - 논리 연산자
    - logical_and, logical_or, logical_not, 등

Functor (Function Object)

```
template<class object>
class less {
    public:
    bool operator() (const object  & lhs, const object  & rhs) const
          {  return lhs < rhs;}
}

int main() {
    …
    itr = find_if (v,begin(), v.end(), less());
    …
    // 지역 객체 l을 선언하고 이 객체를 find_if에 전달해도 됨
    less l;
    itr2 = find_if (v.begin(), v.end(), l);
}
```

Containers

Container	Header file
vector	<vector>
deque	<deque>
list	<list>
set	<set>
multiset	<set>
map	<map>
multimap	<map>
queue	<queue>
priority_queue	<queue>
stack	<stack>

Sequence Containers

- vector
  - 전통적인 형태의 배열 (Array)을 대체
- list
  - 이중 연결 리스트 (Doubly-linked list)의 형태
- deque
  - 벡터와 리스트의 장점을 고루 갖춤. 양 끝 큐 (Doubly-ended queue)의 형태
- 연산
  - 멤버 함수 insert와 erase를 통해 중간에 원소를 삽입/삭제 가능
  - 보통 이들은 list를 제외하고는 비효율적으로 동작하기 때문에 되도록 하지 않는 것이 좋다.
  - 대신 deque에서는 push_back, push_front, pop_back, pop_front 등을, vector에선 push_back, pop_back을 사용하는 것이 좋다.

Vector Container

- 크기 조정이 가능한 동적 배열 (Dynamic array)
- 요소에 random-access 가능
- 처음과 끝에 새로운 요소 삽입 또는 제거 가능
- 자동으로 메모리 관리함

```
vector<int> vList;          // integer 타입의 벡터 리스트 초기화
for (int i = 1; i < 20; i++)
   vList.push_back(i);     // push_back을 사용해서 값을 추가
for (int j = 0; j < vList.size(); j++)
    cout << vList[j] << " ";
cout << endl;
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

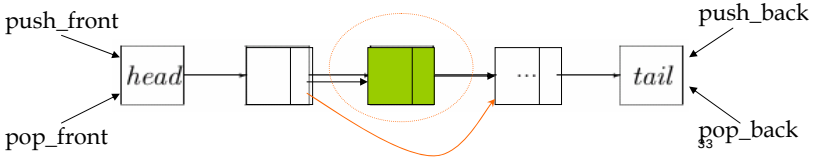
Strings

- 문자의 벡터 (Vector of characters)라고 생각하면 됨
- 상위 개념의 데이터 타입이라고 보면 됨
- String 관련한 연산이 많이 있어 사용이 편리함

```
string aName = "Kyoung Shin Park"
```

List Container

- 이중 연결 리스트 (Doubly-linked list)
- 리스트의 요소는 노드라는 구조체로 관리하며 노드끼리는 링크로 서로 연결되어 있어 요소의 논리적인 순서를 기억
- Bidirectional access – 양쪽 방향으로 traversal 가능
- 리스트가 커지거나 작아질 때 임의의 크기와 메모리를 효과적으로 사용
- 시작과 끝 혹은 중간에서 요소 삽입과 제거 가능
- 리스트는 요소를 자주 추가 혹은 제거할 때, 요소에 대한 random access가 필요하지 않을 때 효과적



List Container

```
#include <list>
list<float> aList;
aList.push_back(0.0);    // 맨 뒤에 0.0 추가
aList.push_front(0.0);   // 맨 앞에 0.0 추가
aList.insert(++aList.begin(), 2.3) // 맨 앞 다음에 2.3 삽입
aList.push_back(5.1);
aList.push_back(6.2);
list<float>::iterator i;
for (i = aList.begin(); i != aList.end(); i++)
    cout << *i << " ";          0.0 2.3 0.0 5.1 6.2
cout << endl;
aList.remove(0.0);              2.3 5.1 6.2
list<float>::iterator found = find(aList.begin(), aList.end(), 5.1);
if (found != aList.end()) cout << *found << endl;
```


Deque (Double Ended Queue) Container

- 벡터 컨테이너와 동일하나 Doubled ended queue
- 필요에 따라 자체적으로 크기 증가와 감소 가능
- front 또는 back end에서 요소 삽입 또는 제거 가능
- 또한, 요소를 특정 위치에 삽입가능
- 요소에 random-access 가능

```
deque<int> dList;          // integer 디큐 리스트 초기화
dList.push_front(2);       // push_front를 사용해서 값을 추가
dList.push_back(4);        // push_back을 사용해서 값을 추가
```

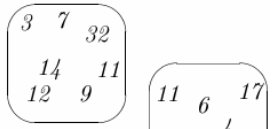
Associative Containers

컨테이너	특성		헤더
set	키가 곧 원소	동일한 값을 갖는 키 허용 안 됨	<set>
multiset		동일한 값을 갖는 키 허용	
map	키와 데이터가 분리	동일한 값을 갖는 키 허용 안 됨	<map>
multimap		동일한 값을 갖는 키 허용	

Set and Multiset Container

- 순서있는 집합
- 효과적인 요소의 삽입, 제거, 포함 함수 지원
- 병합, 합, 차이, 등등 집합관련 연산 함수 지원
- Multiset은 같은 값을 갖는 여러 개 entry를 지원

```
set<string> sData;
sData.insert("apple");
sData.insert("banana");
sData.insert("pear");
set<string>::iterator p = sData.begin();
do {
    cout << *p << " ";
    p++;
} while (p != sData.end());
```



Map (Dictionary) Container

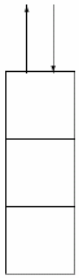
- 키 (Key)와 값 (Value) 쌍의 집합체
- 키 (Key)는 순서있는 데이터 타입이면 가능 (예, string)
- 값 (Value)은 어떤 데이터 타입이건 가능
- 효과적인 요소의 삽입, 제거, 포함 함수 지원
- Multimap 컨테이너는 맵 컨테이너와 비슷하나 duplicate key를 지원

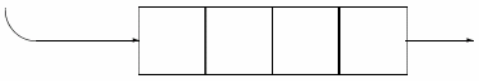
```
map<string, int> mData;
for (int i=0; i<10; i++)
    mData.insert(pair<string><char>('mcb' + i, 100 + i));
```

$$key_1 \rightarrow value_1$$
$$key_2 \rightarrow value_2$$
$$key_3 \rightarrow value_3$$
$$\ldots$$
$$key_n \rightarrow value_n$$

Stack/Queue/Priority-Queue

- 벡터, 리스트, 디큐로 컨테이너 어답터 (Container Adaptor)
- 스택 (Stack)은 Last in First out (LIFO) deque
- 큐 (Queue)는 First in First out (FIFO) deque
- 우선 순위 큐 (Priority-queue)는 vector의 일종





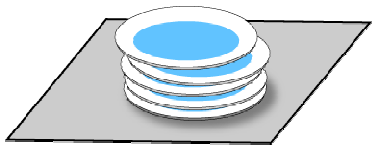
큐

스택

Stack Container Adaptor

- LIFO (Last-In First-Out)
- 큐와 비슷하나 요소를 스택의 Top에 요소를 추가할 수 있고 Top에서 제거 가능 (예, 부엌에 접시 스택)
- Top (맨 위에 위치한 원소 반환), Push (맨 위에 원소를 추가), Pop (맨 위에 위치한 원소 삭제)이 스택의 주요 함수

```
stack<string> stData;
stData.push("apple");
stData.push("banana");
stData.push("pear");
while (!stData.empty()) {
    cout << stData.top() << " ";
    stData.pop();
}
```



Stack Build with Vector Container

```
#include <vector>
#include <stack>
void main()
{
    stack < int, vector<int> > st;
    for (int i=0; i < 10; i++)
        st.push(i);
    while ( !st.empty() )
    {
        cout << st.top() << " ";
        st.pop();
    }
    cout << endl;
}
```


Queue Container Adaptor

- FIFO (First-In First-Out)
- 먼저 들어간 요소가 먼저 나감 (예, 줄서기)
- Front (맨 앞에 위치한 원소 반환), Back (맨 뒤에 위치한 원소 반환), Push (맨 뒤에 원소를 추가), Pop (맨 앞에 원소를 삭제)이 큐의 주요 함수

```
queue<string> quData;
quData.push("apple");
quData.push("banana");
quData.push("pear");
while (!quData.empty()) {
    cout << quData.front() << " ";
    quData.pop();
}
```

Queue Build with List Container

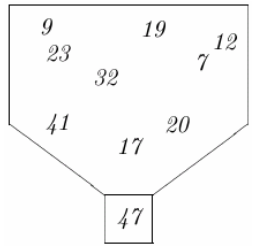
```
#include <list>
#include <queue>
void main()
{
    queue < int, list<int> > qu;
    for (int i=0; i < 10; i++)
        qu.push(i);
    while ( !qu.empty() )
    {
        cout << qu.front() << " ";
        qu.pop();
    }
    cout << endl;
}
```


Priority Queue Container Adaptor

- 기존의 큐는 FIFO방식으로 2 3 1 순서로 넣었으면 2 3 1 순서로 내주지만, 우선 순위 큐 (priority queue)는 2 3 1 순서로 넣었으면 3 2 1 순서로 내준다.
- 우선 순위로 새로운 값을 삽입하는 데 효과적
- 가장 큰 값 또는 가장 작은 값을 접근하는데 효과적

```
priority_queue <int> pqData;
int array[10] = {1, 4, 1, 6, 4, 8, 2, 3, 10, 135};
for (int i = 0; i < 10; i++)
    pqData.push(array[i]);
while ( !pqData.empty() ) {
    cout << pqData.front() << " ";
    pqData.pop();
}
```

135 10 8 6 4 4 3 2 1 1



Operations all Container support

- bool empty() const;
  - 만약 컨테이너가 원소를 가지고 있지 않다면 true, 아니면 false를 반환
- iterator begin() const;
  - 컨테이너에서 시작 위치 iterator를 반환
- iterator end() const;
  - 컨테이너에서 마지막 요소 다음 위치 (next one past the last element) iterator를 반환
- int size() const;
  - 컨테이너에 있는 요소의 개수를 반환