C++ TEMPLATES

Chapter 1. Function Templates

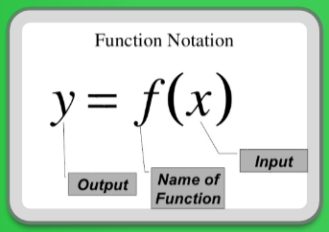
Chapter 2. Class Templates

남 정호

INDEX

The Basics

- Macro & Template
- Standard Template Library
- Template vs. Generic



Function Templates

- Template Parameters
- Argument Deduction & Specification
- Overloading Function Templates

Class Templates

- Implementations
- Specializations
- Overloading Class Templates

1. THE BASICS

1. Macro & Template
2. Standard Template Library
3. Template vs. Generic

1. Macro & Template

- 템플릿이라는 개념은 우연히 발견되었다고 한다.

- 옛날옛적, 템플릿이 없던 시절의 개발자들은
- 매크로를 사용해 제네릭한 객체들을 만듦
- What did people do before templates in C++?

- 매크로 함수
- 매크로 클래스

1. Macro & Template

- 오른쪽과 같이
- 매크로를 복잡하게 사용하다가
- 오늘날의 템플릿을 발견함

- 매크로 함수 -> 템플릿 함수
- 매크로 클래스 -> 템플릿 클래스

```
#define MIN(x, y) (((a)<(b))?(a):(b))
#define MAX(x, y) (((a)>(b))?(a):(b))

#define collection(T) \
class collection_##T \
{ \
private: \
    T *data_; \
    size_t size_; \
    size_t capacity_; \
    \
public: \
    collection_##T() \
        : data_(nullptr), \
        size_(0), capacity_(0) {}; \
    collection_##T(size_t n, const T &val) \
    { \
        data_ = new T[n]; \
        size_ = capacity_ = n; \
        \
        while (n != 0) \
        { \
            *data_++ = val; \
            --n; \
        } \
    }; \
    \
    T* begin() \
    { \
        return data_; \
    }; \
    T* end() \
    { \
        return data_ + size_; \
    }; \
};
```

2. Standard Template Library

- 표준 템플릿 라이브러리
- C++ Templates 를 대표하는 라이브러리

- 크게 네 파트로 나뉨
  - Containers
  - Iterators
  - Algorithms
  - Fuctors

- STL (Standard Template Library) 을 직접 구현하다보면,
- 템플릿에 통달할 수 있지 않을까?

2. Standard Template Library

- 참고자료
- http://www.cplusplus.com/reference/
- http://en.cppreference.com/w/cpp

3. Template vs. Generic

- C++ 의 Template 이,
- 다른 언어 (Java, C#, TypeScript 등) 의 Generic 과
- 어떻게 다른지 알아보자

3. Template vs. Generic

C++ Template

- Macro 에서 진화한 코드 **복제 도구**

- **새로운 타입**을 사용할 때마다
- **새로운 코드**가 생성됨

- 다른 타입의 템플릿 클래스는 이름만 비슷할 뿐,
- 아주 **다른 (타입의) 클래스**이다.

3. Template vs. Generic

C++ Template

```
vector<int> v1;
vector<string> v2;

bool is_same = typeid(v1) == typeid(v2);
cout << is_same << endl; // false
```

3. Template vs. Generic

Generic

- 자동 형 변환을 해 주는 **편의 도구**
- **새로운 타입**을 사용해도
- **동일한 코드**를 사용함

- 다른 타입의 제네릭 클래스는
- 실제 **같은 타입의 클래스**
    - 단지 자동 변환되는 타입만 다를 뿐

3. Template vs. Generic

Generic

```
let v1: std.Vector<number> = new std.Vector();
let v2: std.Vector<string> = new std.Vector();

let is_same: boolean = typeof v1 == typeof v2;

console.log(is_same); // true
```

3. Template vs. Generic

Type	Pros.	Cons.
Template	속도가 빠르며, 유연하다	프로그램의 크기가 커진다
Generic	프로그램의 크기가 작다	속도가 느리며, 유연하지 못다

Template 이 Generic 에 비해

왜 유연하고 어떻게 빠르다는 걸까?

3. Template vs. Generic

```
namespace std
{
    template<typename Key, typename T,
        typename Hash = hash<Key>,
        typename Pred = equal_to<Key>,
        typename Alloc = allocator<pair<const Key, T>>
    > class unordered_map
    {
    public:
        template <typename InputIterator>
        void assign(InputIterator first, InputIterator last);
    };
};
```

3. Template vs. Generic

```
namespace std
{
    class HashMap<Key, T>
    {
        public constructor
        (
            hash_fn: (key: Key) => number = std.hash,
            equal_fn: (x: Key, y: Key) => boolean = std.equal_to
        );

        public assign<Key, T, IForwardIterator<IPair<Key, T>>>
            (first: IForwardIterator, last: IForwardIterator): void;
    }

    interface IForwardIterator<T>
    {
        readonly value: T;
        next(): IForwardIterator<T>;
        equals(obj: IForwardIterator<T>): boolean;
    }
```

3. Template vs. Generic

- Generic 에는 독특한 Dependency 가 존재함

- ArrayContainer -> Vector, Deque 의 부모 클래스
- MapContainer -> 다음 컨테이너들의 조상 클래스
  - TreeMap
  - TreeMultiMap
  - HashMap
  - HashMultiMap

- 다음 페이지의 예제 소스를 보면서
- Template 에 비해 어떤 불편함이 숨어있는지 이야기 해 보자

3. Template vs. Generic

```
6   declare namespace std
7   {
8       export interface IArrayContainer<T> {}
9
10      export abstract class ArrayContainer<T, Source extends IArrayContainer<T>>
11          implements IArrayContainer<T>
12      {
13          public begin(): ArrayIterator<T, Source>;
14          public end(): ArrayIterator<T, Source>;
15
16          public rbegin(): ArrayReverseIterator<T, Source>;
17          public rend(): ArrayReverseIterator<T, Source>;
18      }
19
20      export class ArrayIterator<T, Source extends IArrayContainer<T>>
21      {
22          public source(): Source;
23      }
24      export class ArrayReverseIterator<T, Source extends IArrayContainer<T>>
25          extends ReverseIterator
26          <
27              T, Source,
28              ArrayIterator<T, Source>,
29              ArrayReverseIterator<T, Source>
30          > {}
31  }
```

3. Template vs. Generic

```
35  declare namespace std
36  {
37      export interface IMapContainer<Key, T> {}
38
39      export abstract class MapContainer<Key, T, Source extends IMapContainer<Key, T>>
40      {
41          public begin(): MapIterator<Key, T, Source>;
42          public end(): MapIterator<Key, T, Source>;
43
44          public rbegin(): MapReverseIterator<Key, T, Source>;
45          public rend(): MapReverseIterator<Key, T, Source>;
46      }
47
48      export class MapIterator<Key, T, Source extends IMapContainer<Key, T>>
49          implements IMapContainer<Key, T>
50      {
51          public source(): MapContainer<Key, T, Source>;
52      }
53      export class MapReverseIterator<Key, T, Source extends IMapContainer<Key, T>>
54          extends ReverseIterator
55          <
56              Entry<Key, T>,
57              MapContainer<Key, T, Source>,
58              MapIterator<Key, T, Source>,
59              MapReverseIterator<Key, T, Source>
60          > {}
61  }
```

2. FUNCTION TEMPLATES

1. Template Parameters
2. Argument Deductions
3. Overloading Function Templates

1. Template Parameters

```
template <typename T>
T max(T x, T y)
{
    return x > y ? x : y;
}
```

- 하나의 템플릿 파라미터

- 서로 다른 타입의 파라미터를
- 사용하고자 할 때는?

1. Template Parameters

```
template <typename X, typename Y>
X max(X x, Y y)
{
    return x > y ? x : y;
};

template <typename X, typename Y, typename Ret>
Ret max(X x, Y y)
{
    return x > y ? x : y;
};
```

- 서로 다른 두 개 타입의
- 대소를 비교할 때,

- X 타입이 리턴되어야 하는가?
- Y 타입이 리턴되어야 하는가?

2. Argument Deductions

```
template <typename X, typename Y>
auto max(X x, Y y);

template <typename X, typename Y>
auto max(X x, Y y) -> decltype(a > b ? a : b);

template <typename X, typename Y>
auto max(X x, Y y) -> std::decay<decltype(a > b ? a : b)>::type;

template <typename X, typename Y>
auto max(X x, Y y) -> std::common_type_t<X, Y>;

template <typename X, typename Y,
          typename Ret = std::decay<decltype(a > b ? a : b)>::type>
auto max(X x, Y y) -> Ret;
```

2. Argument Deductions

- 리턴 타입을 컴파일러에 맡김

```
template <typename X, typename Y>
auto max(X x, Y y);
```

- 리턴 타입을 로직을 통해 추론함

```
template <typename X, typename Y>
auto max(X x, Y y) -> decltype(a > b ? a : b);
```

2. Argument Deductions

- 리턴 타입을 로직을 통해 추론하되, 원형을 유지함

```
template <typename X, typename Y>
auto max(X x, Y y) -> std::decay<decltype(a > b ? a : b)>::type;
```

- std::decay 가 하는 일은?

```
std::decay<int>::type; // int
std::decay<int&>::type; // int
std::decay<int&&>::type; // int
std::decay<const int>::type; // int
std::decay<const int&>::type; // int
std::decay<const int&&>::type; // int
```

2. Argument Deductions

- 리턴 타입을 decay 를 default 로,
- 하지만 사용자가 원하거든 변경할 수 있도록

```
template <typename X, typename Y,
          typename Ret = std::decay<decltype(a > b ? a : b)>::type>
auto max(X x, Y y) -> Ret;
```

3. Overloadings

```
template<>
short max(short x, short y)
{
    return (x + (~y + 1) >> 15 == 0) ? x : y;
};

template<>
int max(int x, int y)
{
    return (x + (~y + 1) >> 31 == 0) ? x : y;
};

template <typename X, typename Y, typename Z>
auto max(X x, Y y, Z z)
{
    return max(x, max(y, z));
};
```

3. Overloadings

```
namespace mystl
{
    template <typename T>
    std::string to_string(T);

    template<> std::string to_string(short val);
    template<> std::string to_string(int val);
    template<> std::string to_string(double val);
};
```

mystl::to_string(**1.3f**) -> 결과는?

3. CLASS TEMPLATES

1. Implementations
2. Specializations
3. Overloading Class Templates

1. Implementations

- 클래스에 Template Argument 를 사용.
- 클래스 내 멤버에 Generic 한 타입을 사용할 수 있다.

- Variables
- Methods
- Type Definitions

- 클래스 템플릿의 가장 대표적인 활용사례
- **Containers**

1. Implementations

```
namespace std
{
    template <typename T, typename Alloc = allocator<T>>
    class vector;

    template <typename T, typename Alloc = allocator<T>>
    class deque;

    template <typename T, typename Alloc = allocator<T>>
    class list;

    template <typename T, typename Alloc = allocator<T>>
    class forward_list;
};
```

1. Implementations

```
namespace std
{
    template
    <
        typename Key,
        typename T,
        typename Comp = less<Key>,
        typename Alloc = allocator<pair<const Key, T>>
    > class map; // TREE-MAP

    template
    <
        typename Key,
        typename T,
        typename Hash = hash<Key>,
        typename Pred = equal_to<Key>,
        typename Alloc = allocator<pair<const Key, T>>
    > class unordered_map; // HASH-MAP
};
```

1. Implementations

- 책에서 간단한 Stack 을 구현함
- 기저 container 는 std::vector

- Template Argument "T" 가
- 클래스 내에서 다양하게 쓰이고 있음

- Argument of Member Variable
- Parameter & Return Type of Methods

```
namespace mystl
{
    template <typename T>
    class stack
    {
    private:
        std::vector<T> elems_;

    public:
        bool empty() const
        {
            return elems_.empty();
        };
        const T& top() const
        {
            return elems_.back();
        };

        void push(const T &elem)
        {
            elems_.push_back(elem);
        };
        void pop()
        {
            elems_.pop_back();
        };
    };
};
```

2. Specializations

- 두 번째 예제 stack<std::string>
  - string 에 한해서
  - 별도로 정의된 클래스를 사용

Quiz

- 기저 컨테이너로 std::vector 가 아닌
- std::deque 를 사용하면
- 어떤 이득이 있을까?

```
namespace mystl
{
    template <>
    class stack<std::string>
    {
    private:
        std::deque<std::string> elems_;

    public:
        bool empty() const;
        const std::string& top() const;

        void push(const std::string &elem);
        void pop();
    };
}
```

2. Specializations

```
namespace std
{
    template <typename T, typename Alloc = allocator<T>>
    class deque
    {
    private:
        vector<vector<T>> data_;
        size_t size_;
        size_t capacity_;

    public:
        void pop_front();
        void push_front(const T &elem);
    };
};
```

2. Specializations

```
namespace mystl
{
    template <typename T, typename Container = std::vector<T>>
    class stack
    {
    private:
        Container elems_;

    public:
        bool empty() const;
        const T& top() const;

        void push(const T &elem);
        void pop();
    };
};
```

2. Specializations

- std::vector<bool>
- STL 에 볼 수 있는 Specialization 중 가장 대표적인 사례

- 왜 std::vector<bool> 은 따로이 정의되었을까?
- bool 의 특징을 생각해보자

```
namespace std
{
    // BOOL SPECIALIZATION
    template <typename Alloc = allocator<bool>>
    class vector<bool, Alloc>;
};
```

3. Overloading Class Templates

- 템플릿 클래스를 overload 한 경우엔,
- std::tuple 이 가장 대표적인 사례

- tuple.hpp

Q & A

2017.01.20

남 정호