Back to Basic) Templates in C++

---

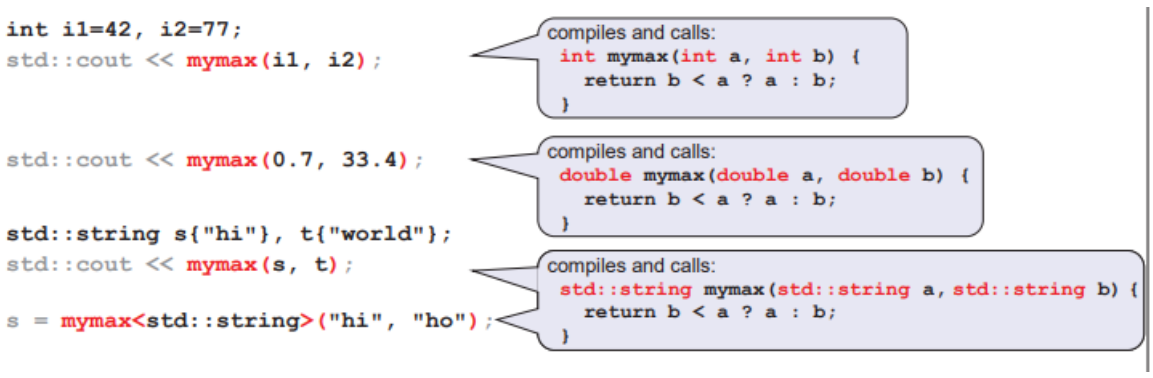
template이란??

- 여러 데이터 타입에 대해 재사용 가능한 코드를 작성할 수 있도록 하는 기능을 의미한다.
- 함수 템플릿과 클래스 템플릿으로 나누어진다.

Function Template

```
template<typename T>
T mymax(T a, T b)
{
        return b < a ? a : b;
}
```

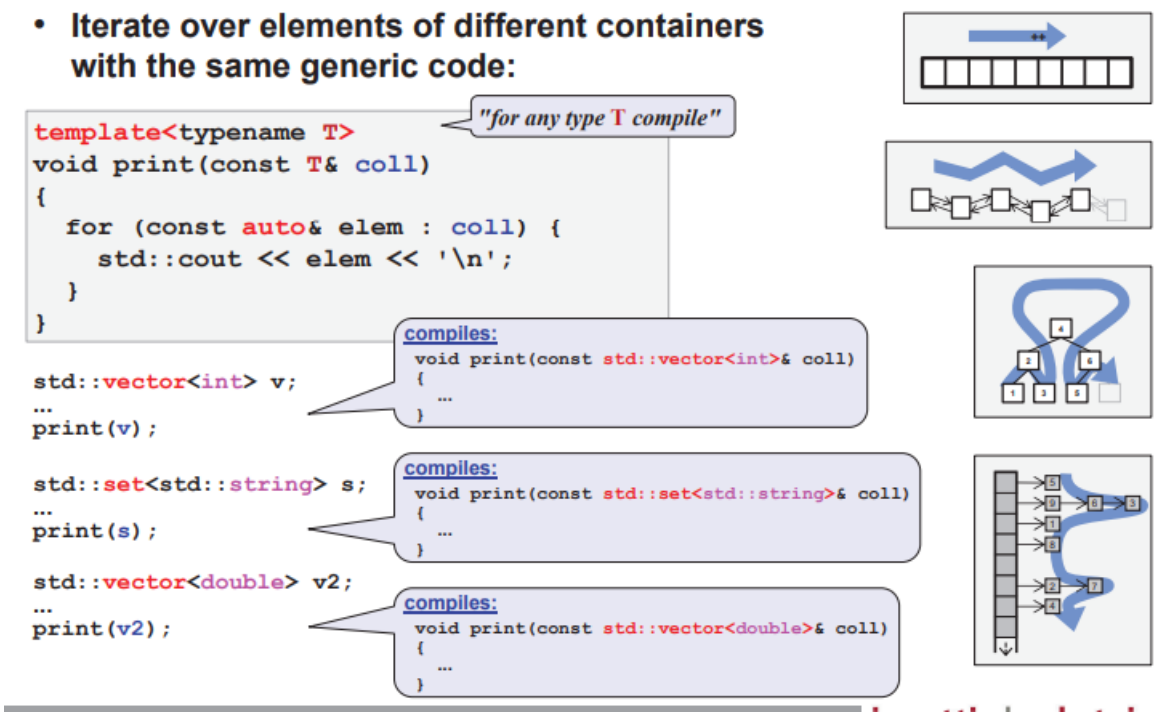
- T는 comman place이다. 그러나 어떤 자료형도 가능하다.
- 컴파일러는 호출하는 argument를 기반으로 **템플릿 인스턴스화**를 진행해서 특정 데이터 타입에 대한 실제 코드를 생성한다.



- 위의 예시를 보면, template instantiation 중, **명시적 인스턴스화**와 **암시적 인스턴스화**의 예시를 알 수 있다.

Generic Iterating

```
template<typename T>
for(const auto& elem: coll)
        std::cout << elem << "\n";
```



만약 특정 container에 들어있는 객체를 출력하고 싶다면, template를 사용해서 더 쉽게 작성할 수 있다.

Templates in Header Files

- **Templates are usually *defined* in header files**
  - Not only *declared*
  - No `inline` necessary

mycode.hpp:

```
template<typename T>
T mymax(T a, T b)
{
  return b < a ? a : b;
}
```

```
#include "mycode.hpp"
…

int i1=42, i2=77;
auto a = mymax(i1, i2);      // OK
auto b = mymax(0.7, 33.4);   // OK
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t);        // OK
```

mycode.hpp:

```
template<typename T>
T mymax(T a, T b);
```

```
#include "mycode.hpp"
…

int i1=42, i2=77;
auto a = mymax(i1, i2);      // Error
auto b = mymax(0.7, 33.4);   // Error
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t);        // Error
```

- Template들은 주로 `header file` 안에 정의되어 있다.
- `inline` 을 넣을 필요는 없다.
- 만약 선언만 하고, 정의를 하지 않는다면 컴파일러가 `링크 오류` 를 발생시킨다.

Function Template Requirements

```
template<typename T>
T mymax(T a, T b)
{
        return b < a ? a : b;
}
```

- 만약 Template 함수를 작성하기 위해서는, 함수에 사용되는 `operation` 이 필요하다.
- 위의 예시를 살펴본다면 2개의 operation이 필요하다. 2개의 object를 비교할 수 있는 연산자가 필요할 것이고, 복사 연산자가 필요하다.

```
int i1=42, i2=77;
std::cout << mymax(i1, i2);

std::cout << mymax(7, 33.4);            // ERROR: can't deduce T (int or double)
std::cout << mymax<double>(7, 33.4);    // OK, T is double

std::complex<double> c1, c2;
std::cout << mymax(c1, c2);             // ERROR: deduces T as complex<>, but no < supported

std::atomic<int> a1{8}, a2{15};
std::cout << mymax(a1, a2);             // ERROR: deduces T as atomic<>, but copying disabled
```

OK, compiles and calls:

```
int mymax(int a, int b) {
  return b < a ? a : b;
}
```

template은 좋은 기능이기는 하지만, 많은 문제점이 있었다. 만약 위의 코드처럼 짧은 함수가 아닌, 1000줄짜리 함수를 작성한다고 한다면, coder가 어떤 requirement를 충족해야하는 지 명시적으로 알 방법이 없다.
이런 불편함을 해결하기 위해서 C++20에서는 `concepts` 를 통해 명시적으로 requirement를 알려준다.

Concepts(Since C++20)

- **Concepts:** (since C++20)
  - To formulate formal constraints for generic code
  - To disable bad behavior or find errors early

```
template<typename T>
concept HasLessThan = requires (T x) { {x < x} -> std::convertible_to<bool>; };

template<typename T>
requires std::copyable<T> && HasLessThan<T>
T mymax(T a, T b)
{
  return b < a ? a : b;
}

int i1=42, i2=77;
std::cout << mymax(i1, i2);

std::complex<double> c1, c2;
std::cout << mymax(c1, c2);             // ERROR: concept HasLessThan not supported

std::atomic<int> a1{8}, a2{15};
std::cout << mymax(a1, a2);             // ERROR: concept std::copyable not supported
```

Concept: Named requirements

Explicit constraint for T:
- copy/move constructor
- operator < returning bool

- Template에 필요한 requirement를 알려주기 위한 것.

Multitple Template Parameters

```
template<typename T1, typename T2>
void print(const T1& val1, const T2& val2)
{
        cout << val1 << " " << val2 << '\n';
}
```

- 템플릿 함수에는 복수의 template parameter를 전달할 수 있다.
- 2개의 template parameter는 같은 자료형도 허용되고, 다른 자료형도 허용된다.

```
int i1=42, i2=77;
print(i1, i2);                  // OK, print<int, int>()
print(0.7, 33.4);               // OK, print<double, double>()
print(i1, 0.7);                 // OK, print<int, double>()

std::string s{"hi"}, t{"world"};
print(s, t);                    // OK, print<std::string, std::string>()
print("hi", "world");           // OK, print<char[3], char[6]>()
print("hi", s);                 // OK, print<char[3], std::string>()

print<double>(i1, i2);          // OK, print<double, int>()
print<double, double>(i1, i2);  // OK, print<double, double>()
```

How to deduce/specify the return type??

```
template<typename T1, typename T2>
??? mymax(T1 a, T2 b)
{
  return b < a ? a : b;
}
```

- Parameters might have different types now

그렇다면 앞에서 살펴보았던 mymax를 살펴보자. 그렇다면 다른 type의 parameter를 템플릿 함수에 넣는다면 어떤 type을 return해야하는가??

c++에서는 모든 expression은 type을 가지고 있다. 이 타입은 `컴파일 타임`에 정의가 된다. C++내부적으로는 이런 상황일 경우 (int와 double을 비교하는 경우) "common type"을 정의하도록 되어있다.

common type이란, C++에서 서로 다른 두 타입을 비교하거나 연산할 때, 컴파일러는 두 타입의 공용 타입을 계산하여 결과값을 결정한다. 즉, 두 타입 중 더 `넓은 표현 범위`를 가진 타입으로 변환(implicit conversion)을 하여 연산을 수행한다.

예를 들어서, int와 double을 비교할 때, int는 double로 변환되어 연산이 이루어지며, 그 결과는 double 타입이 된다.

그렇다면 함수의 return type은 어떻게 설정해야 해야할까?? 왜냐면 우리는 어떤 type이 공용타입인지 모르기 때문이다.

⭐ 정답은 `auto`를 사용하는 것이다. `auto`를 사용하게 되면, 컴파일러가 타입을 자동적으로 추론해준다.

- **Let the compiler deduce the return type**

```
template<typename T1, typename T2>
auto mymax(T1 a, T2 b)
{
  return b < a ? a : b;
}

int i = 42;
std::string s{"hi"};
...

auto a1 = mymax(i, 0.7);          // OK, return type is double
auto a2 = mymax(0.7, i);          // OK, return type is double
auto a3 = mymax<double>(i, 0.7); // OK, return type is double
auto a4 = mymax<long>(0.7, i);    // return type is long (may convert 0.7 to 0)

auto s1 = mymax("hi", s);         // OK, return type is std::string
auto s2 = mymax(s, "world");      // OK, return type is std::string
```

- Let the compiler deduce the return type
  - Since C++14
- Templates are usually a good application of return type `auto`
  - The type is always right
  - Better than guessing wrong