인트아이 C++ 심화 스터디

C++ TEMPLATE, CRTP

인하대학교 인트아이

4회차 퀴즈

답안

```
T pop() {
    std::lock_guard<std::mutex> lock(mutex);
    const T value = queue.front(); (1) Queue 맨 앞의 값을 미리 저장하고
    queue.pop(); (2) 맨 앞의 값을 제거하고
    return value; (3) 저장해둔 값을 출력, lock은 자동으로 풀림
}
```

Template [템플릿]

C 언어에는 없는 C++ 만의 특별한 문법

다양한 타입에 대한 코드를 여러 개 만들지 않아도 됨

컴파일 시간에 타입에 맞는 코드를 생성(코드를 찍어내는 틀) → 다형성 구현 가능

상속을 통해 구현한 다형성은 런타임(프로그램 실행 시간)에 필요한 코드를 불러오기 때문에 느림

템플릿을 통해 구현한 다형성은 컴파일 시간에 필요한 코드가 완성된 상태이기 때문에 빠름

Polymorphism (다형성)

poly(여러 개의) + morphism(형태)

같은 형태의 코드가 다르게 동작하도록 하는 것

다형성은 타입을 일반화해서 코드를 다룰 수 있게 하기 때문에 깔끔한 코드를 작성하기 위해서 중요함

- ✓ 동적(런타임) 다형성은 상속을 통해 구현
- ✓ 정적(컴파일 타임) 다형성은 템플릿을 통해 구현

```
class Animal {
public:
    virtual void print() const {
        std::cout << "Animal\n";
    }
};

class Cat : public Animal {
    void print() const override {
        std::cout << "Cat\n";
    }
};

class Dog : public Animal {
    void print() const override {
        std::cout << "Dog\n";
    }
};

int main() {
    std::vector<Animal *> v;
    v.push_back(new Cat);
    v.push_back(new Dog);
    v[0]->print();
    v[1]->print();
    return 0;
}
```

```
Cat
Dog
```

Template

장점

빠르게 동작하는 코드를 작성할 수 있다.

단점

읽고 쓰기 어렵다. 디버깅이 어렵다.

컴파일 시간이 늘어난다.

프로그램의 크기가 커진다.

정의와 구현을 분리할 수 없고, 모두 헤더파일에 들어간다.

Template = 성능 원툴, 남용하면 코드 꼬이기 딱 좋음, 프로그래밍계의 흑마법

Template

```
template<typename T>
T sum(T a, T b) {
    return a + b;
}

sum<int>(1, 2); // 3
sum<float>(1.1, 2.2); // 3.3
```


```
// 함수 이름은 예시일 뿐이며,
// 실제로 함수 이름이 이렇게 생성되지 않음
int sum_int(int a, int b) {
    return a + b;
}

float sum_float(float a, float b) {
    return a + b;
}

sum_int(1, 2); // 3
sum_float(1.1, 2.2); // 3.3
```

sum 템플릿 함수는 sum_int와 sum_float를 찍어내는 틀이 되어 각 타입에 맞는 새로운 코드를 생성

template<typename T>은 과거에 template<class T> 형태로 작성되었으나,
class가 주는 어감(ex. int, char 등은 클래스가 아니지만 T에 들어올 수 있음)으로 인해 typename이 추가되었습니다.
class와 typename은 이름만 다를 뿐 완전히 동일한 의미의 키워드로, typename 사용을 권장합니다.

Template

```
template<typename T = int>
struct Complex {
    T real;
    T imaginary;
};
```

함수 뿐만 아니라 구조체, 클래스 등에도 template을 사용할 수 있음

함수 인자와 마찬가지로 템플릿 인자도 기본 인자(default parameter)를 가질 수 있음

Template Specialization (템플릿 특수화)

```
struct Complex {
    int real;
    int imaginary;
};

template<typename T>
T sum(T a, T b) {
    return a + b;
}

template<>
Complex sum(Complex a, Complex b) {
    return Complex{
        a.real + b.real,
        a.imaginary + b.imaginary
    };
}

sum(1, 2); // 3
sum(1.1, 2.2); // 3.3
sum(Complex{ 1, 2 }, Complex{ 3, 4 }); // Complex { 4, 6 }
```

Complex 구조체에 대해서는 template〈typename T〉 sum 템플릿이 아닌 아래의 template〈〉 sum 함수가 호출

이처럼 특정한 타입에 대해 템플릿 코드를 선택하는 것을 템플릿 특수화라 부름

sum(1, 2)을 sum〈int〉(1, 2)으로 쓰지 않아도 되는 이유는 함수 인자 1에서 int라는 정보를 추론할 수 있기 때문

STL의 Template Specialization

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;                                          (1)

namespace pmr {
    template< class T >
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;   (2) (since C++17)
}
```

Specializations

The standard library provides a specialization of `std::vector` for the type `bool`, which may be optimized for space efficiency.

`vector<bool>`	space-efficient dynamic bitset (class template specialization)

std::vector〈bool〉은 템플릿 특수화의 대표적인 예시

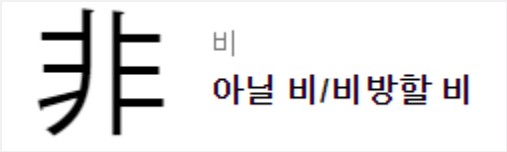
`std::vector<bool>` behaves similarly to `std::vector`, but in order to be space efficient, it:

- Does not necessarily store its elements as a contiguous array.
- Exposes class `std::vector<bool>::reference` as a method of accessing individual bits. In particular, objects of this class are returned by `operator[]` by value.
- Does not use `std::allocator_traits::construct` to construct bit values.
- Does not guarantee that different elements in the same container can be modified concurrently by different threads.

bool은 일반적인 배열이 아닌 2진수의 각 비트를 이용해 값을 저장

비(非)타입(non-type) 템플릿 인자

템플릿 인자로 typename 이외의 것도 전달 가능




```
template<int i>
int bonus(int n) {
    return n + i;
}

bonus<100>(10); // 110
```

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```

(since C++11)

```
// construction uses aggregate initialization
std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 prior to
                                    // the CWG 1270 revision (not needed in C++11
                                    // after the revision and in C++14 and beyond)

std::array<int, 3> a2 = {1, 2, 3};  // double braces never required after =

std::array<std::string, 2> a3 = { std::string("a"), "b" };
```

std::array가 비타입 템플릿 인자를 사용하는 대표적인 예시

CRTP

Curiously Recurring Template Pattern

: 기묘한 재귀 템플릿 패턴?

스스로를 인자로 받는 템플릿을 상속하는 클래스

virtual을 사용하지 않고 클래스의 메소드를 재사용하는 기법

```
template<class T>
class Base {
};

class Derived : public Base<Derived> {
};
```

curious ★★

1. 형용사 궁금한, 호기심이 많은 (=inquisitive)
2. 형용사 별난, 특이한, 기이한

recur

동사 되풀이 되다, 다시 일어나다

CRTP

```
template<typename T>
class Animal {
public:
    void print() const {
        (static_cast<const T &>(*this)).print();
    }
};

class Cat : public Animal<Cat> {
public:
    void print() const {
        std::cout << "Cat\n";
    }
};

template<typename T>
void print_animal(const Animal<T> &animal) {
    animal.print();
}
```

```
Cat
```

virtual 없이 다형성을 구현할 수 있음

→ 정적 다형성 (Static Polymorphism)

성능 손실 없이 class마다 동일한 메소드 구현이 필요할 때 사용

Non-virtual class

```
class Animal {
public:
    void print() const { // non-virtual
        std::cout << "Animal\n";
    }
};

class Cat : public Animal {
    void print() const {
        std::cout << "Cat\n";
    }
};

void print_animal(const Animal &animal) {
    animal.print();
}
```

```
int main() {
    Cat cat;
    print_animal(cat);
    return 0;
}
```

Animal

virtual이 없으면 의도한대로 다형성이 구현되지 않음

Quiz

주어진 코드를 참고하여 main함수가 동작하도록 move_up, move_down, move_left를 구현하시오.

```cpp
#include "quiz.h"

int main() {
    Cat cat(1, 2);
    print_animal(cat); // (1,2)
    move_right(cat);
    print_animal(cat); // (2,2)
    move_up(cat);
    print_animal(cat); // (2,3)
    move_left(cat);
    print_animal(cat); // (1,3)
    move_down(cat);
    print_animal(cat); // (1,2)
    return 0;
}
```

```cpp
#include <iostream>

template<typename T>
class Animal {
public:
    Animal(int x, int y)
    : x(x), y(y) {}
    void move(int x, int y) {
        this->x = x;
        this->y = y;
    }
    void print() const {
        (static_cast<const T &>(*this)).print();
    }
    int get_x() const {
        return x;
    }
    int get_y() const {
        return y;
    }
private:
    int x;
    int y;
};

class Cat : public Animal<Cat> {
public:
    Cat(int x, int y)
    : Animal(x, y) {}
    void print() const {
        std::cout << "Cat: " << get_x() << ", " << get_y() << "\n";
    }
};

template<typename T>
void print_animal(const Animal<T> &animal) {
    animal.print();
}

template<typename T>
void move_right(Animal<T> &animal) {
    animal.move(animal.get_x() + 1, animal.get_y());
}
```