인트아이 C++ 심화 스터디

SFINAE, CONSTRAINTS & CONCEPTS

인하대학교 인트아이

5회차 퀴즈

답안

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

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

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

```
Cat: 1, 2
Cat: 2, 2
Cat: 2, 3
Cat: 1, 3
Cat: 1, 2
```

SFINAE [스피내]

Substitution Failure Is Not An Error(치환 실패는 오류가 아님)

컴파일러가 잘못된 일부 템플릿 코드를 무시(오류x)할 수 있도록 기능

컴파일러가 상황에 따라 유효한 템플릿을 선택 → 템플릿에서의 조건문 구현

템플릿 특수화보다 더 복합적인 조건에 대해서 대응가능

(ex. 그냥 typename T가 아니라, T가 print 메소드를 가진 클래스 타입일 때만 선택되게)

CRTP와 마찬가지로, 정적 다형성 구현에 이용

CRTP 복습

```
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

SFINAE 원리

타입(using)을 포함하는 빈 구조체(struct)를 정의해서 사용

```cpp
struct Box {
    using inner_t = int; // `inner_t`를 정의, `Box::inner_t`는 `int`
};

template<typename T>
void print(typename T::inner_t t) { // 만약, `T`에 `inner_t`가 정의되어 있다면 이 함수를 오버로딩
    // `t`의 타입은 `T::inner_t`
    std::cout << "inner_t is defined: " << t << std::endl;
}

template<typename T>
void print(T t) { // 만약, `T`에 `inner_t`가 정의되어 있지 않다면 이 함수를 오버로딩
    std::cout << "inner_t is not defined: " << t << std::endl;
}

int main() {
    print<Box>(10);
    print<double>(20);
    return 0;
}
```

```
inner_t is defined: 10
inner_t is not defined: 20
```

Template Meta Function [템플릿 메타 함수]

메타 함수 : 값이 아닌 타입에 대해 연산하는 함수

템플릿 메타 함수 : 템플릿을 통해 구현한 타입에 대해 연산하는 함수

템플릿 함수이기 때문에 func(a,b) 형태가 아닌 func<a, b> 형태로 사용


```cpp
template<typename T, typename U>
struct is_same {
    static constexpr bool value = false;
};

template<typename T>
struct is_same<T, T> {
    static constexpr bool value = true;
};

template<typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;

int main() {
    std::cout << is_same<int, int>::value << '\n';
    std::cout << is_same_v<int, double> << '\n';
    return 0;
}
```

constexpr는 컴파일 시간에 값이 확정되는 코드임을 의미

::value를 template을 이용해 생략하기 때문에 메타 함수는 *_v 형태의 이름을 가짐

```
1
0
```

is_same<int, double>::value 와 동일

SFINAE :enable_if

템플릿 메타 함수의 출력 결과에 따라 템플릿 사용을 결정하는 메타 함수

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
    using type = T;
};

template<bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

```
#include <type_traits>

int main() {
    std::cout << std::is_integral_v<int> << std::endl
              << std::is_integral_v<float> << std::endl;
    return 0;
}
```

```
1
0
```

타입이 정수형(int, long, char, bool) 이면 true 반환

`std::enable_if_t<std::is_integral_v<T>>` 형태로 사용

SFINAE :enable_if

```cpp
template<typename T>
struct Point {
    std::string name;
    T x;
    T y;
};

template<typename T>
typename std::enable_if_t<std::is_integral_v<T>> print_point(const Point<T> &point) { // `T`가 정수형이면 이 함수를 오버로딩
    std::cout << point.name << '<' << typeid(T).name() << '>' << " = integral point (" << point.x << ", " << point.y << ")" << std::endl;
}

template<typename T>
typename std::enable_if_t<!std::is_integral_v<T>> print_point(const Point<T> &point) { // `T`가 정수형이 아니면 이 함수를 오버로딩
    std::cout << point.name << '<' << typeid(T).name() << '>' << " = non-integral point (" << point.x << ", " << point.y << ")" << std::endl;
}

int main() {
    Point<int> p0{ "p0", 1, 2 };
    Point<long long> p1{ "p1", 3ll, 4ll };
    Point<float> p2{ "p2", 0.1f, 0.2f };
    Point<double> p3{ "p3", 0.3, 0.4 };
    print_point(p0);
    print_point(p1);
    print_point(p2);
    print_point(p3);
    return 0;
}
```

```
p0<int> = integral point (1, 2)
p1<__int64> = integral point (3, 4)
p2<float> = non-integral point (0.1, 0.2)
p3<double> = non-integral point (0.3, 0.4)
```

아이디어는 좋은데
“너무” 어렵다

Constraints & Concepts

C++20에서 추가, 타입에 제약조건을 걸 수 있는 새로운 문법

```
template<typename T>
struct Point {
    std::string name;
    T x;
    T y;
};

template<typename T>
    requires std::is_integral_v<T>  T가 is_integral_v를 만족해야 함
void print_point(const Point<T> &point) { // `T`가 정수형이면 이 함수를 오버로딩
    std::cout << point.name << '<' << typeid(T).name() << '>' << " = integral point (" << point.x << ", "
<< point.y << ")" << std::endl;
}

template<typename T>
void print_point(const Point<T> &point) { // `T`가 정수형이 아니면 이 함수를 오버로딩
     std::cout << point.name << '<' << typeid(T).name() << '>' << " = non-integral point (" << point.x << ", "
<< point.y << ")" << std::endl;
}
```

```
p0<int> = integral point (1, 2)
p1<__int64> = integral point (3, 4)
p2<float> = non-integral point (0.1, 0.2)
p3<double> = non-integral point (0.3, 0.4)
```

Constraints & Concepts

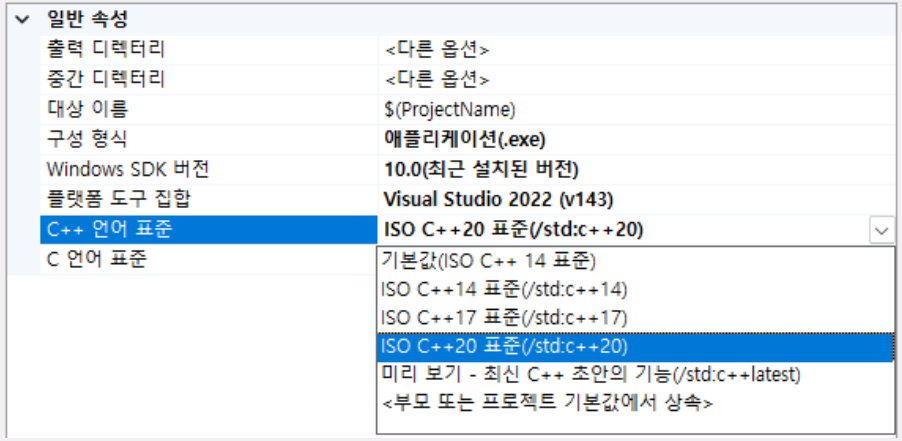
```
template<typename T>
concept Subtractable = requires(T a, T b) { a - b; }; { } 안의 식이 컴파일 가능한지 확인, concept = 제약조건이 걸린 type

template<Subtractable T> typename 대신에 concept 사용
void subtract_value(const T &a, const T &b) {
    std::cout << a - b << '\n';
}

template<typename T> 첫번째 템플릿 조건을 만족하지 않으면 두번째 템플릿 적용
void subtract_value(const T &a, const T &b) {
    std::cout << a << "-" << b << '\n’;
}

int main() {
    int i = 10;
    int j = 5;
    std::string s = "aaa";
    std::string t = "bb";
    subtract_value(i, j);
    subtract_value(s, t);
    return 0;
}
```

```
5
aaa-bb
```



Quiz

(1) T가 print() 메소드를 가지고 있는지 검사하는 Printable Concept를 정의하고,

(2) print 메소드가 존재하는 타입만을 인자로 받는 print_obj 함수를 구현하시오.

```cpp
class Tree {
public:
    Tree(int x, int y)
    : x(x), y(y) {}

    void print() const {
        std::cout << "Tree: " << get_x() << ", " << get_y() << "\n";
    }

    int get_x() const {
        return x;
    }

    int get_y() const {
        return y;
    }

private:
    int x;
    int y;
};

template<typename T>
concept Printable = /* */;

template</* */>
void print_obj(/* */) {
    obj.print();
}
```

```cpp
#include "quiz.h"

int main() {
    Cat cat(1, 2);
    print_obj(cat); // (1,2)
    move_right(cat);
    print_obj(cat); // (2,2)
    move_up(cat);
    print_obj(cat); // (2,3)
    move_left(cat);
    print_obj(cat); // (1,3)
    move_down(cat);
    print_obj(cat); // (1,2)

    Tree tree(0, 3);
    print_obj(tree); // (0,3)
    return 0;
}
```

```
Cat: 1, 2
Cat: 2, 2
Cat: 2, 3
Cat: 1, 3
Cat: 1, 2
Tree: 0, 3
```

Cat 클래스는 5회차 퀴즈 코드와 동일

Hint

다 풀기 전까지는 되도록이면 페이지를 넘기지 말 것

```cpp
template<typename T>
concept Printable = requires(T obj) { obj.print(); };
```