함수 템플릿

일반화 프로그래밍 (generic programming)

C++의 특징 중 하나는 일반화 프로그래밍(generic programming)이다. 일반화 프로그래밍은 데이터를 중시하는 객체 지향 프로그래밍과는 달리 프로그램의 알고리즘에 중점을 둔다. 이러한 일반화 프로그래밍을 지원하는 C++의 대표적인 기능 중 하나가 바로 템플릿(template)이다.

템플릿이란?

템플릿은 매개변수의 타입에 따라 함수나 클래스를 생성하는 메커니즘을 의미하며, 타입이 매개변수에 의해 표현된다. 템플릿을 사용하면 타입마다 별도의 함수나 클래스를 만들지 않고, 여러 타입에서 동작할 수 있는 단 하나의 함수나 클래스를 작성하는 것이 가능해진다.

함수 템플릿

함수 템플릿은 함수의 일반화된 선언을 의미한다. 함수 템플릿을 사용하면 같은 알고리즘을 기반으로, 서로 다른 타입에서 동작하는 함수를 한 번에 정의할 수 있다. 임의의 타입으로 작성된 함수에 특정 타입을 매개변수로 전달하면, 컴파일러는 해당 타입에 맞는 함수를 생성해준다.

함수 템플릿은 아래와 같은 문법으로 정의한다.

```
template <typename 타입 이름>
함수의 원형 {
    // function
}
```

예제

아래 예제는 변수의 값을 서로 바꿔주는 swap()함수를 함수 템플릿을 이용하여 작성한 코드이다. string 변수 이외의 타입에도 정상적으로 작동한다.

```cpp
#include <iostream>
using namespace std;
template <typename DTYPE> void swap(DTYPE &x, DTYPE &y);

int main() {
    string cherry = "cherry", peach = "peach";
    cout << "Befor swap" << endl;
    cout << "fruit1 : " << cherry  << ", fruit2 : " << peach << "\n" << endl;
    swap(cherry, peach);
    cout << "After swap" << endl;
    cout << "fruit1 : " << cherry  << ", fruit2 : " << peach << endl;

    return 0;
}

template <typename DTYPE>
void swap(DTYPE &x, DTYPE &y) {
    DTYPE tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
#include <iostream>
using namespace std;
template <typename DTYPE> void swap(DTYPE &x, DTYPE &y);

int main() {
    string cherry = "cherry", peach = "peach";
    cout << "Befor swap" << endl;
    cout << "fruit1 : " << cherry  << ", fruit2 : " << peach << "₩n" << endl;
    swap(cherry, peach);
    cout << "After swap" << endl;
    cout << "fruit1 : " << cherry  << ", fruit2 : " << peach << endl;

    return 0;
}

template <typename DTYPE>
void swap(DTYPE &x, DTYPE &y) {
    DTYPE tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

함수 템플릿의 인스턴스화

함수 템플릿이 각각의 타입에 대해 처음으로 호출될 때, 컴파일러는 해당 타입의 인스턴스를 생성한다다. 이렇게 생성된 인스턴스는 해당 타입에 대해 특수화된 템플릿 함수이다. 이 인스턴스는 함수 템플릿에 해당 타입이 사용될 때마다 호출된다.

클래스 템플릿

클래스 템플릿이란?

클래스 템플릿은 클래스의 일반화된 선언을 의미한다. 위에서 살펴본 함수 템플릿과 동작은 같으며, 그 대상이 클래스라는 점만 다르다. 클래스 템플릿을 사용하면 타입에 따라 다르게 동작하는 클래스 집합을 만들 수 있다는 장점이 있다. 즉, 클래스 템플릿에 전달되는 템플릿 인수(template argument)에 따라 별도의 클래스를 만들 수 있게 된다.

클래스 템플릿은 다음과 같은 문법으로 정의할 수 있다.

```
template <typename 타입이름>
class 클래스템플릿이름 {
    // class
}
```

예제

아래 예제는 클래스 템플릿을 이용하여 여러 타입의 데이터를 저장할 수 있는 코드이다.

```
#include <iostream>
using namespace std;

template <typename DTYPE> class Data {
    private:
        DTYPE data;

    public:
        Data(DTYPE info);
        DTYPE get_data();
};
```

```
int main() {
	Data<string> str_data("C++");
	Data<int> int_data(10);

	cout << "str_data : " << str_data.get_data() << endl;
	cout << "int_data : " << int_data.get_data() << endl;

	return 0;
}


template <typename DTYPE> Data<DTYPE>::Data(DTYPE info) {
	data = info;
}


template <typename DTYPE> DTYPE Data<DTYPE>::get_data() {
	return data;
}
```

클래스 템플릿의 특징

1. 하나 이상의 템플릿 인수를 가지는 클래스 템플릿을 선언할 수 있다.
2. 클래스 템플릿에 디폴트 템플릿 인수를 명시할 수 있다.
3. 클래스 템플릿를 기초 클래스로 하여 상속할 수 있다.