Template Meta Programming

20200670 이시헌

1. Template?

- 일반화 프로그래밍을 위한 기능.
- 타입에 독립적인 코드 작성을 가능케 하는 도구.
- 다양한 언어에서 템플릿 기능을 지원한다.
- C++에서는 다음과 같이 작성.
  - ```
    template <typename T>
    T add(T a, T b) { return a + b; }
    ```
    ```

2. TMP

- 템플릿을 사용하여 작성된,
  컴파일 시간에 동작하는 프로그램을 작성하는 것.
  (컴파일러가 실행하는 프로그램을 작성하는 것)

3. Pros of TMP

1. 일부 까다롭거나 처리 불가능한 작업을 손쉽게 해결.
2. 작업을 런타임 영역에서 컴파일 타임 영역으로 전환.

전환하여 얻는 이익

1. 일부 런타임 에러를 프로그램 실행 이전에 감지 가능.
2. 실행 코드 감소, 실행 시간 단축, 메모리 소모 감소.
   (컴파일 타임은 그만큼 증가한다)

4. How to use?

- 추후 설명할 방법으로, 직접 TMP 코드를 작성할 수 있다.
  하지만 그런 방법은 범용 고수준 언어 대신, 어셈블리어를 직접 작성하는 것과 유사하다.
  그러므로 이미 구현된 TMP 라이브러리를 사용해 보기를 권장한다. (C++의 경우는 Boost -> MPL)
- 이미 많은 라이브러리가 TMP를 활용하고 있다.
  즉, 당신은 이미 TMP로 작성된 코드를 사용해 보았다!

5.1. Type parameters

- 우리가 잘 아는 대부분의 템플릿 변수는
  Type parameter로 쓰이고 있다.
- 앞서 살펴본 간단한 add 함수에 사용된 T또한
  특정한 타입의 자료를 입력 받는
  Type parameter a와 b를 선언하는데 사용되었다.
  - ```
    template <typename T>
    T add(T a, T b) { return a + b; }
    ```
    ```

5.2. Non-type parameters

- 특이한 점은,

  우리가 상수를 매개변수로 사용할 수 있다는 것.

- 정수, 포인터 등의 상수를 템플릿 변수에 전달하는 예.

  - ```
    Template<int N>
    struct Factorial {
      static const int value = N * Factorial<N-1>::value;
    };
    ```
  - ```
    std::cout << Factorial<10>::value;
    ```

5.2. Non-type parameters

- 즉, 컴파일러가 코드 내부에서,
  템플릿 변수를 상수로 대체한다는 것.
- 상수가 템플릿 객체에 넘기는 자료형 처럼 동작한다는 뜻.
- #define의 동작 방식과 동일.
  (컴파일러의 기계적 코드 번역.)

6. Code Preview

- Iteration
- Factorial
- Fibonacci

7. constexpr keyword

- Constant expressions 의 약자이다. (상수 표현)
- 앞서, 직접 수행하는 TMP가,
  어셈블리어 작성과 유사하다고 하였다.
- C++11 이상에서 지원하는 constexpr 키워드가 TMP를
  간편하게 적용할 수 있도록 돕는다.
- 함수와 클래스에 모두 적용 가능하다.
  (C++14 부터 함수 내부의 if, for 등의 기능 허용,
  C++20부터 함수 내부의 동적 메모리 할당 허용)

7.1. difference of constexpr and TMP

- 다만, 템플릿 함수는 컴파일타임에 수행되기 위하여, 모든 매개변수가 상수여야만 한다.
- constexpr는 분명 런타임에 동작하므로,
  일반 변수를 전달받을 수 있다.
  (템플릿 함수를 직접 작성하는 것에 비하여 제약이 적다.)

7.2. constexpr with Function

```
constexpr int add(int a, int b) {
    return a + b;
}

constexpr int result = add(3, 5);
```

7.3. constexpr with Class

```
struct Point {
    int x, y;
    constexpr Point(int a, int b) : x(a), y(b) {}
};

constexpr Point p(1, 2);
```

7.4. constexpr with Factorial

```
constexpr int factorial(int n) {
    int result = 1;
    for (int i = 1; i ≤ n; ++i) {
        result *= i;
    }
    return result;
}

std::cout << factorial(10);
```

8. 참고자료

- Meyers, Scott. EFFECTIVE C++ (3rd Edition). P.332-346
- ‘cppreference.com’. Constant expressions. ‘https://en.cppreference.com/w/cpp/language/constant_expression’.

Q&A

감사합니다.