8. 다형성

◈ 다형성의 개념(Polymorphism)

- 동일한 인터페이스로 여러 가지 동작을 수행하는 성질
  - 인터페이스 – 함수 이름, 연산자 이름, 변수 이름 등

◈ 다형성의 특징

- 동일한 함수 또는 연산자 이름을 사용함으로써 코드를 간결하게 사용할 수 있음
- 코드 분석 시 포인터 또는 형변환을 통한 다형성은 이해하기 어려움
  : 기본클래스 포인터로 여러 개의 파생클래스에 접근이 가능함
- 설계 단계에서 다형성을 활용하고자 하는 클래스, 함수, 연산자 등에 대해 고려해야 함

◈ 다형성의 종류

- 애드 혹 다형성 (Ad-hoc polymorphism)
  : 전달 인자 (argument)에 따라 다르게 행동하는 방식
  ex) 오버로딩 (overloading) – 함수, 연산자, 템플릿

- 매개 변수 다형성 (Parametric polymorphism)
  : 범용적 데이터 타입으로 동일하게 동작하는 방식
  ex) 템플릿 (template)

- 서브 타입 다형성 (Subtype polymorphism)
  : 실행 시점에 동작을 결정하는 방식
  ex) 동적 바인딩 (dynamic binding or late binding)

제5장 템플릿(Template)

포함과 상속은 코드를 재활용하는 좋은 방법이지만 자료형에 대한 수정이 필요한 경우는 재사용할 수 없다. 예를 들어, 정수값을 저장하는 클래스를 실수값을 저장하는 용도로 사용하려면 반드시 자료형을 수정해야 한다. 이 때 템플릿을 사용하면 일반형으로 프로그램을 작성한 후에 원하는 자료형에 따라 코드를 자동으로 생성할 수 있다.

1. Template 함수

- 함수 템플릿은 알고리즘은 같으나 자료형이 다른 경우 각 자료형에 맞는 동일한 함수를 빠르고 정확하게 만들어낼 때 사용
- 함수템플릿은 함수를 정의하는 것이 아니며 단지 함수를 만드는 방법을 기술하는 것이다.
- 컴파일러는 함수의 호출문장에 사용된 자료형을 보고 적절한 실제 함수의 정의부를 만들어낸다. 결국 컴파일 후에는 자료형에 따라 각각의 함수가 별도로 만들어지는 셈이다.

[예제5-1] template 함수

(temp.h)

```
#pragma once
#include<cstring>
template <typename Type>  // 함수템플릿의  정의
Type max(Type &a, Type &b){
        if(a>=b) return a;
        else return b;
}
template <>
char *max(char *&a, char *&b) { // 템플릿 특수화(전문화) 정의
        if(strcmp(a, b)>0) return a;
        else return b;
}
```

(main.cpp)

```
#include <iostream>
using namespace std;
#include "temp.h"
int main(){
        int a=10, b=20, ires;
        double x=2.4, y=3.5, dres;
        char *ap="apple";
        char *bp="banana";
        char *resp;

        ires=max(a, b);  // int형 값으로 호출
        cout << "두 정수 중에서 큰 값 : " << ires << endl;
        dres=max(x, y);  // double형 값으로 호출
        cout << "두 실수 중에서 큰 값 : " << dres << endl;
        resp=max(ap, bp);
        cout << "두 문자열 중에서 큰 문자열 : " << resp << endl;

        return 0;
}
```

2. Template class

- 클래스 또한 템플릿을 만들 수 있다.
- 일반적인 자료형으로 클래스 템플릿을 만들고 필요할 때 특정 자료형의 클래스가 만들어지도록 하면 빠르고 정확하게 다양한 자료형의 클래스를 구현할 수 있다.
- 클래스 템플릿은 특정 자료형의 객체를 생성할 때 클래스의 정의가 완성된다.

[예제5-2] template class

(temp.h)

```
// 클래스 템플릿 정의
template <typename Type>  // 형 매개변수는 Type
class Temp{
 private:
        Type num;
 public:
        Temp(Type n);  // 생성자
        void put(Type n);  // 값 저장
        Type get();  // 값 확인
};

// 멤버함수 템플릿, 해당 클래스를 지정할 때 형 매개변수를 사용한다.
template <typename Type>
Temp<Type>::Temp(Type n)
{
        num=n;
}

template <typename Type>
void Temp<Type>::put(Type n)
{
        num=n;
}
template <typename Type>
Type Temp<Type>::get()
{
        return num;
}
```

(main.cpp)

```
#include <iostream>
using namespace std;
#include "temp.h"

int main()
{
        Temp<int> a(10);  // int형의 Temp클래스를 만들고 객체 생성
        Temp<double> b(3.5);  // double형 Temp를 만들고 객체 생성

        cout << a.get() << endl;
        b.put(2.4);
        cout << b.get() << endl;

        return 0;
}
```

3. 템플릿 특수화

템플릿 인수를 특정 타입으로 고정하는 가능
특정 타입이 사용될 때 원본 템플릿 대신 특수화 템플릿이 사용 됨
특수화 템플릿의 이름은 원본 템플릿의 이름과 동일하게 사용 하나 내부의 구현은 다를 수 있음.

(예)

```
//원본 템플릿
template<typename T>
Class Obj {
    void Foo() { cout << __FUNCTION__ << endl;
};

//char* 형에 대한 특수화 템플릿
template<>   ← 범용데이터 타입명을 지정하지 않음
class Obj<char*> {
    void Foo() { cout << "Specialization for type char*" << endl;
};
```

[예제5-3] 템플릿 특수화

(temp.h)

```
template <typename T1, typename T2>
class Pair {                    // 원본 템플릿

        T1 first;
        T2 second;
public:
        Pair(T1 d1, T2 d2) : first(d1), second(d2) {}
        void Show() { cout << "origin pair(" << first << ", " << second << ")"
<< endl; }
};

template <typename T1>
class Pair <T1, const char *> {   // 부분 특수화 템플릿

        T1 first;
        char * second;
public:
        Pair(T1 d1, const char * d2) : first(d1) {
                int len = strlen(d2) +1;
                second = new char[len];
                memcpy(second, d2, len);
```

```
        }
        ~Pair() { delete[] second; }
        void Show() { cout << "part pair(" << first << ", " << second << ")" <<
endl; }
};

template <>
class Pair <const char*, const char *> {   // 특수화 템플릿
        char * first;
        char * second;
public:
        Pair(const char * d1, const char * d2 ) {
                int len = strlen(d1)+1;
                first = new char[len];
                memcpy(first, d1, len);
                len = strlen(d2) +1;
                second = new char[len];
                memcpy(second, d2, len);
        }
        ~Pair() {
                delete[] first;
                delete[] second;
        }
        void Show() { cout << "all pair(" << first << ", " << second << ")" <<
endl;  }
};
```

(main.cpp)

```
#include <iostream>
using namespace std;
#include "temp.h"

int main()
{
//… 호출
    const char* id = "foo";
    const char* passwd = "1234567";
    Pair<double, double> mydata1(3.1456, 1.73205);
    Pair<int, const char *> mydata2(127, passwd);
    Pair<const char*, const char *> mydata3(id, passwd);
    mydata1.Show();
    mydata2.Show();
    mydata3.Show();

    return 0;
}
```

4. 템플릿 적용 스마트 포인터

연산자 오버로딩을 이용해서 만든 스마트 포인터 코드에 템플릿 개념을 적용해서 활용해 보자

[예제5-4] 템플릿 적용 스마트 포인터

(ISmartPointer.h)

```
#ifndef _ISmartPointer_H
#define _ISmartPointer_H
template<typename T>
struct Isp
{
        virtual ~Isp() {};
        virtual T * operator->() const = 0;
        virtual T * operator&() const = 0;
        virtual T & operator*() const = 0;
};
#endif
```

(my_sp.h)

```
#ifndef _MY_SP_H
#define _MY_SP_H
#include "ISmartPointer.h"
template<typename T>
class my_sp : public Isp<T> {
public:
        my_sp(T * p) : ptr(p) {}
        virtual ~my_sp() { delete ptr; }
        T * operator->() const  { return ptr; }
        T * operator&() const { return ptr; }
        T & operator*() const { return *ptr; }
private:
        T * ptr;
};
#endif
```

(light.h)

```
#ifndef _LIGHT_H
#define _LIGHT_H

#include <iostream>
#include "my_sp.h"
using namespace std;
class Light {
public:
```

```
        Light() : _stat(false) { cout << "call " << __FUNCTION__ << endl; }
        ~Light() { cout << "call " << __FUNCTION__ << endl; }
        void setOn() { _stat = true; }
        void setOff() { _stat = false; }
        bool getStat() { return _stat; }
private:
        bool _stat;
};
inline void room_ctl(my_sp<Light>& r, bool on = true) {
        on ? r->setOn() : r->setOff();
}
#endif
```

(main.cpp)

```
#include <iostream>
using namespace std;
#include “light.h"

int main()
{
        my_sp<Light> room1(new Light);
        my_sp<Light> room2(new Light);
        my_sp<Light> room3(new Light);
        room_ctl(room1, false);
        room_ctl(room2);
        room_ctl(room3, false);
        cout << boolalpha << "room1 : " << room1->getStat() << endl;
        cout << "room2 : " << room2->getStat() << endl;
        cout << "room3 : " << room3->getStat() << endl;
        return 0;
}
```

* 위의 예제를 smart pointer 라이브러리로 변경해서 실행해보자