템플릿(Template) 2

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

SeoulTech

Point 클래스 템플릿과 배열 클래스 템플릿

```
template <typename T>
class BoundCheckArray
{
private:
    T * arr;
    int arrlen;
    BoundCheckArray(const BoundCheckArray& arr) { }
    BoundCheckArray& operator=(const BoundCheckArray& arr) { }
public:
    BoundCheckArray(int len);
    T& operator[] (int idx);
    T operator[] (int idx) const;
    int GetArrLen() const;
    ~BoundCheckArray();
};
```

일반적인 배열 클래스 템플릿

```
template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
};
```

```
BoundCheckArray<int> iarr(50);
```
int형 데이터 저장 배열

```
BoundCheckArray<Point<int>> oarr(50);
```
Point<int> 객체 저장 배열

```
BoundCheckArray<Point<int>*> oparr(50);
```
Point<int>형 포인터 저장 배열

```
typedef Point<int>* POINT_PTR;
BoundCheckArray<POINT_PTR> oparr(50);
```
Point<int>형 포인터 저장 배열

템플릿 클래스 대상의 함수선언과 friend 선언

```
template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0): xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
    friend Point<int> operator+(const Point<int>&, const Point<int>&);
    friend ostream& operator<<(ostream& os, const Point<int>& pos);
};
Point<int> operator+(const Point<int>& pos1, const Point<int>& pos2)
{
    return Point<int>(pos1.xpos+pos2.xpos, pos1.ypos+pos2.ypos);
}
```

컴파일러가 생성해 내는 템플릿 클래스를 함수의 매개변수 및 반환형으로 지정하는 것도 가능하고 이러한 함수를 대상으로 friend 선언을 하는 것도 가능하다.

결론은 컴파일러가 생성하는 템플릿 클래스의 이름도 일반 자료형의 이름과 차별을 받지 않는다는 것!

PointTemplate.h

```
#ifndef __POINT_TEMPLATE_H_
#define __POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
};

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y)
{ }

template <typename T>
void Point<T>::ShowPosition() const
{
    cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
}

#endif
```

ArrayTemplate.h

```
#ifndef __ARRAY_TEMPLATE_H_
#define __ARRAY_TEMPLATE_H_

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

template <typename T>
class BoundCheckArray
{
private:
	T * arr;
	int arrlen;

	BoundCheckArray(const BoundCheckArray& arr) { }
	BoundCheckArray& operator=(const BoundCheckArray& arr) { }

public:
	BoundCheckArray(int len);
	T& operator[] (int idx);
	T operator[] (int idx) const;
	int GetArrLen() const;
	~BoundCheckArray();
};

template <typename T>
BoundCheckArray<T>::BoundCheckArray(int len) :arrlen(len)
{
	arr=new T[len];
}

template <typename T>
T& BoundCheckArray<T>::operator[] (int idx)
{
	if(idx<0 || idx>=arrlen)
	{
		cout<<"Array index out of bound exception"<<endl;
		exit(1);
	}
	return arr[idx];
}
template <typename T>
T BoundCheckArray<T>::operator[] (int idx) const
{
	if(idx<0 || idx>=arrlen)
	{
		cout<<"Array index out of bound exception"<<endl;
		exit(1);
	}
	return arr[idx];
}
template <typename T>
int BoundCheckArray<T>::GetArrLen() const
{
	return arrlen;
}
template <typename T>
BoundCheckArray<T>::~BoundCheckArray()
{
	delete []arr;
}
#endif
```

BoundArrayMain.cpp

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

int main(void)
{
	BoundCheckArray<Point<int>> oarr1(3);
	oarr1[0]=Point<int>(3, 4);
	oarr1[1]=Point<int>(5, 6);
	oarr1[2]=Point<int>(7, 8);

	for(int i=0; i<oarr1.GetArrLen(); i++)
		oarr1[i].ShowPosition();

	BoundCheckArray<Point<double>> oarr2(3);
	oarr2[0]=Point<double>(3.14, 4.31);
	oarr2[1]=Point<double>(5.09, 6.07);
	oarr2[2]=Point<double>(7.82, 8.54);

	for(int i=0; i<oarr2.GetArrLen(); i++)
		oarr2[i].ShowPosition();

	typedef Point<int>*		POINT_PTR;
	BoundCheckArray<POINT_PTR> oparr(3);
	oparr[0]=new Point<int>(11, 12);
	oparr[1]=new Point<int>(13, 14);
	oparr[2]=new Point<int>(15, 16);

	for(int i=0; i<oparr.GetArrLen(); i++)
		oparr[i]->ShowPosition();

	delete oparr[0]; delete oparr[1]; delete oparr[2];

	return 0;
}
```

PointTemplateFriendFunction.cpp

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

template <typename T>
class Point
{
private:
	T xpos, ypos;
public:
	Point(T x=0, T y=0): xpos(x), ypos(y)
	{ }
	void ShowPosition() const
	{
		cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
	}

	friend Point<int> operator+(const Point<int>&, const
	Point<int>&);
	friend ostream& operator<<(ostream& os, const Point<int>&
	pos);
};

Point<int> operator+(const Point<int>& pos1, const Point<int>&
	pos2)
{
	return Point<int>(pos1.xpos+pos2.xpos, pos1.ypos+pos2.ypos);
}

ostream& operator<<(ostream& os, const Point<int>& pos)
{
	os<<'['<<pos.xpos<<", "<<pos.ypos<<']'<<endl;
	return os;
}

int main(void)
{
	Point<int> pos1(2, 4);
	Point<int> pos2(4, 8);
	Point<int> pos3=pos1+pos2;
	cout<<pos1<<pos2<<pos3;
	return 0;
}
```

클래스 템플릿 특수화

```
template <typename T>
class SoSimple
{
public:
    T SimpleFunc(T num) { . . . . }
};
```

```
template <>
class SoSimple<int>
{
public:
    int SimpleFunc(int num) { . . . . }
};
```

SoSimple 클래스 템플릿에 대해서 int형에 대한 특수화

√ 클래스 템플릿을 특수화하는 이유는 특정 자료형을 기반으로 생성된 객체에 대해, 구분이 되는 다른 행동 양식을 적용하기 위함이다.

√ 함수 템플릿을 특수화하는 방법과 이유, 그리고 클래스 템플릿을 특수화하는 방법과 이유는 동일하다.

ClassTemplateSpecializaion.cpp

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

template <typename T>
class Point
{
private:
	T xpos, ypos;
public:
	Point(T x=0, T y=0): xpos(x), ypos(y)
	{ }
	void ShowPosition() const
	{
		cout<<'['<<xpos<<",
	"<<ypos<<']'<<endl;
	}
};

template <typename T>
class SimpleDataWrapper
{
private:
	T mdata;
public:
	SimpleDataWrapper(T data) : mdata(data)
	{ }
	void ShowDataInfo(void)
	{
		cout<<"Data: "<<mdata<<endl;
	}
};
```

```
template<>
class SimpleDataWrapper <char*>
{
private:
	char* mdata;
public:
	SimpleDataWrapper(char* data)
	{
		mdata=new char[strlen(data)+1];
		strcpy(mdata, data);
	}
	void ShowDataInfo(void)
	{
		cout<<"String: "<<mdata<<endl;
		cout<<"Length:
	"<<strlen(mdata)<<endl;
	}
	~SimpleDataWrapper()
	{
		delete []mdata;
	}
};
```

```
template<>
class SimpleDataWrapper <Point<int>>
{
private:
		Point<int> mdata;
public:
		SimpleDataWrapper(int x, int
y) : mdata(x, y)
		{ }
		void ShowDataInfo(void)
		{

		mdata.ShowPosition();
		}
};

int main(void)
{
		SimpleDataWrapper<int>
iwrap(170);
		iwrap.ShowDataInfo();

		SimpleDataWrapper<char*>
swrap("Class Template Specialization");
		swrap.ShowDataInfo();

		SimpleDataWrapper<Point<int
>> poswrap(3, 7);
		poswrap.ShowDataInfo();
		return 0;
}
```

클래스 템플릿의 부분 특수화

```
template <typename T1, typename T2>
class MySimple { . . . . }
```

MySimple 클래스 템플릿

```
template <>
class MySimple<char, int> { . . . . }
```

MySimple 클래스 템플릿의 <char, int>에 대한 특수화

```
template <typename T1>
class MySimple<T1, int> { . . . . }
```

MySimple 클래스 템플릿의 <T1, int>에 대한 부분적 특수화

T2가 int 인 경우에는 MySimpe<T1, int>를 대상으로 인스턴스가 생성된다.

위와 같이 <char, int>형으로 특수화, 그리고 <T1, int> 에 대해서 부분 특수화가 모두 진행된 경우 특수화가 부분 특수화에 앞선다. 즉, <char, int>를 대상으로 객체 생성시 특수화된 클래스의 객체가 생성된다.

ClassTemplatePartialSpecialization.cpp

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

template <typename T1, typename T2>
class MySimple
{
public:
	void WhoAreYou()
	{
		cout<<"size of T1: "<<sizeof(T1)<<endl;
		cout<<"size of T2: "<<sizeof(T2)<<endl;
		cout<<"<typename T1, typename T2>"<<endl;
	}
};

template<>
class MySimple<int, double>
{
public:
	void WhoAreYou()
	{
		cout<<"size of int: "<<sizeof(int)<<endl;
		cout<<"size of double: "<<sizeof(double)<<endl;
		cout<<"<int, double>"<<endl;
	}
};

template<typename T1>
class MySimple<T1, double>
{
public:
	void WhoAreYou()
	{
		cout<<"size of T1: "<<sizeof(T1)<<endl;
		cout<<"size of double: "<<sizeof(double)<<endl;
		cout<<"<T1, double>"<<endl;
	}
};


int main(void)
{
	MySimple<char, double> obj1;
	obj1.WhoAreYou();
	MySimple<int, long> obj2;
	obj2.WhoAreYou();
	MySimple<int, double> obj3;
	obj3.WhoAreYou();
	return 0;
}
```

템플릿 매개변수에는 변수의 선언이 올 수 있습니다.

```
template <typename T, int len>
class SimpleArray
{
private:
    T arr[len];
public:
    T& operator[] (int idx)
    {
        return arr[idx];
    }
};
```

템플릿의 인자로 변수의 선언이 올 수도 있다!

```
class SimpleArray<int, 5>
{
private:
    int arr[5];
public:
    int& operator[] (int idx) { return arr[idx]; }
};
```

SimpleArray<int, 5> i5arr;

SimpleArray<int, 5>형 템플릿 클래스

```
class SimpleArray<double, 7>
{
private:
    double arr[7];
public:
    double& operator[] (int idx) { return arr[idx]; }
};
```

SimpleArray<double, 7> i7arr;

SimpleArray<double, 7>형 템플릿 클래스

템플릿 인자를 통해서 SimpleArray<int, 5>와 SimpleArray<int ,7>이 서로 다른 자료형으로 인식되게 할 수 있다.

이로써 SimpleArray<int, 5>와 SimpleArray<int, 7> 사이에서의 연계성을 완전히 제거할 수 있다.

```
int main(void)
{
    SimpleArray<int, 5> i5arr1;
    SimpleArray<int, 7> i7arr1;
    i5arr1=i7arr1;    // 컴파일 Error!
    . . . . .
}
```

NonTypeTemplateParam.cpp

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

template <typename T, int len>
class SimpleArray
{
private:
    T arr[len];
public:

    T& operator[] (int idx)
    {
            return arr[idx];
    }
    T& operator=(const T&ref)
    {
            for(int i=0; i<len; i++)
                    arr[i]=ref.arr[i];
    }
};
```

```
int main(void)
{
    SimpleArray<int, 5> i5arr1;
    for(int i=0; i<5; i++)
            i5arr1[i]=i*10;

    SimpleArray<int, 5> i5arr2;
    i5arr2=i5arr1;
    for(int i=0; i<5; i++)
            cout<<i5arr2[i]<<", ";
    cout<<endl;

    SimpleArray<int, 7> i7arr1;
    for(int i=0; i<7; i++)
            i7arr1[i]=i*10;

    SimpleArray<int, 7> i7arr2;
    i7arr2=i7arr1;
    for(int i=0; i<7; i++)
            cout<<i7arr2[i]<<", ";
    cout<<endl;
    return 0;
}
```

템플릿 매개변수는 디폴트 값 지정도 가능합니다.

```
template <typename T=int, int len=7>
class SimpleArray
{
private:
    T arr[len];
public:
    T& operator[] (int idx) { return arr[idx]; }
    SimpleArray<T, len>& operator=(const SimpleArray<T, len> &ref)
    {
        for(int i=0; i<len; i++)
            arr[i]=ref.arr[i];
    }
};
```

디폴트 값 지정 가능!

```
int main(void)
{
    SimpleArray<> arr;
    for(int i=0; i<7; i++)
        arr[i]=i+1;
    for(int i=0; i<7; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
    return 0;
}
```

T에 int, len에 7의 디폴트 값 지정!

실행결과

```
1 2 3 4 5 6 7
```

함수 템플릿과 static 지역변수

```
template <typename T>
void ShowStaticValue(void)
{
    static T num=0;
    num+=1;
    cout<<num<<" ";
}
```

```
void ShowStaticValue<int>(void)
{
    static int num=0;
    num+=1;
    cout<<num<<" ";
}
```

```
void ShowStaticValue<long>(void)
{
    static long num=0;
    num+=1;
    cout<<num<<" ";
}
```

함수 템플릿의 static 변수는 템플릿 함수 별로 독립적이다!

```
int main(void)
{
    ShowStaticValue<int>();
    ShowStaticValue<int>();
    ShowStaticValue<int>();
    cout<<endl;
    ShowStaticValue<long>();
    ShowStaticValue<long>();
    ShowStaticValue<long>();
    cout<<endl;
    ShowStaticValue<double>();
    ShowStaticValue<double>();
    ShowStaticValue<double>();
    return 0;
}
```

```
1 2 3
1 2 3
1 2 3
```

실행결과

TemplateParamDefaultValue.cpp

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

template <typename T=int, int len=7>
class SimpleArray
{
private:
	T arr[len];
public:

	T& operator[] (int idx)
	{
		return arr[idx];
	}
	T& operator=(const T&ref)
	{
		for(int i=0; i<len; i++)
			arr[i]=ref.arr[i];
	}
};

int main(void)
{
	SimpleArray<> arr;
	for(int i=0; i<7; i++)
		arr[i]=i+1;
	for(int i=0; i<7; i++)
		cout<<arr[i]<<" ";
	cout<<endl;
	return 0;
}
```

클래스 템플릿과 static 멤버변수

```
template <typename T>
class SimpleStaticMem
{
private:
    static T mem;
public:
    void AddMem(int num) { mem+=num; }
    void ShowMem() { cout<<mem<<endl; }
} ;
template <typename T>
T SimpleStaticMem<T>::mem=0;      // 이는 템플릿 기반의 static 멤버 초기화 문장이다.
```

클래스 템플릿의 static 변수는 템플릿 클래스 별로 독립적이다! 따라서 템플릿 클래스 별 객체들 사이에서만 공유가 이뤄진다.

```
class SimpleStaticMem<int>
{
private:
    static int mem;
public:
    void AddMem(int num) { mem+=num; }
    void ShowMem() { cout<<mem<<endl; }
} ;
int SimpleStaticMem<int>::mem=0;
```

SimpleStaticMem<int>의 mem은
SimpleStaticMem<int>의 개체간 공유

```
class SimpleStaticMem<double>
{
private:
    static double mem;
public:
    void AddMem(double num) { mem+=num; }
    void ShowMem() { cout<<mem<<endl; }
} ;
double SimpleStaticMem<double>::mem=0;
```

SimpleStaticMem<double>의 mem은
SimpleStaticMem<int>의 개체간 공유

FunctionTemplateStaticVar.cpp

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

template <typename T>
void ShowStaticValue(void)
{
	static T num=0;
	num+=1;
	cout<<num<<" ";
}

int main(void)
{
	ShowStaticValue<int>();
	ShowStaticValue<int>();
	ShowStaticValue<int>();
	cout<<endl;

	ShowStaticValue<long>();
	ShowStaticValue<long>();
	ShowStaticValue<long>();
	cout<<endl;

	ShowStaticValue<double>();
	ShowStaticValue<double>();
	ShowStaticValue<double>();
	return 0;
}
```

template<typename T> vs. template<>

```
template <typename T>
class SoSimple
{
public:
    T SimpleFunc(T num) { . . . . }
};
```

템플릿임을 알리며 T가 무엇인지에 대한 설명도 필요한 상황

```
template <typename T>
T SimpleStaticMem<T>::mem=0;
```

```
template <>
class SoSimple<int>
{
public:
    int SimpleFunc(int num) { . . . . }
};
```

템플릿과 관련 있음을 알리기만 하면 되는 상황

```
template <>
long SimpleStaticMem<long>::mem=5;
```

static 멤버 초기화의 특수화

ClassTemplateStaticMem.cpp

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

template <typename T>
class SimpleStaticMem
{
private:
    static T mem;

public:
    void AddMem(int num) { mem+=num; }
    void ShowMem() { cout<<mem<<endl; }
} ;

template <typename T>
T SimpleStaticMem<T>::mem=0;

/*
template<>
long SimpleStaticMem<long>::mem=5;
*/
```

```
int main(void)
{
    SimpleStaticMem<int> obj1;
    SimpleStaticMem<int> obj2;
    obj1.AddMem(2);
    obj2.AddMem(3);
    obj1.ShowMem();

    SimpleStaticMem<long> obj3;
    SimpleStaticMem<long> obj4;
    obj3.AddMem(100);
    obj4.ShowMem();
    return 0 ;
}
```

Q & A