Programming Study
C++ Template

Sungwoo Nam

2018.3.21

Parameterized Type

```
template<typename T>
class Vector {
private:
      T* elem;
      int sz;

public:
      explicit Vector(int s) {
            elem = new T[s];
            sz = s;
      }

      T& operator[](int i)
      {
            if (i < 0 || i >= size())
                        throw out_of_range("[i] is out of bound");
            return elem[i];
      }
};
```

Parameterized Type - Usage

```
{
	Vector<int> vi(200);
	vi[0] = 123;
	assert(vi[0] == 123);
	CatchExceptionMessage<out_of_range>([&]{ vi[200] = 321; });
}

{
	Vector<string> vs(17);
	vs[3] = "Hello";
	assert(vs[3] == "Hello");
	CatchExceptionMessage<out_of_range>([&]{ string x = vs[-1]; });
}

{
	Vector<vector<int>> vli(45);
	vli[4] = vector<int>({ 1, 2, 3 });
	assert(vli[4][1] == 2);
	CatchExceptionMessage<out_of_range>([&]{ vli[45][3]; });
}
```

Function Template

```
template<typename T>
class Vector {
...

      T* begin() const
      {
            return size() ? elem : nullptr;
      }

      T* end() const
      {
            return begin() + size();
      }
};

template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
      for (auto x : c)
            v += x;
      return v;
}
```

Function Template - Usage

```
{
	Vector<int> vi(4);
	vi[0] = 0;
	vi[1] = 1;
	vi[2] = 2;
	vi[3] = 3;

	double ds = sum(vi, 0.0);
	assert(ds == 6.0);

	int is = sum(vi, 0);
	assert(is == 6);
}

{
	list<double> ld;
	ld.push_back(3.0);
	ld.push_back(4.0);

	double ds = sum(ld, 0.0);
	assert(ds == 7.0);
}
```

Function Object

```
class Shape
{
public:
	virtual void draw() = 0;
	virtual void rotate( double degree ) = 0;
};

class Rect : public Shape
{
public:
	void draw() { cout << "rect"; }
	void rotate(double degree) { cout << "rotate"; }
};

class Circle : public Shape
{
public:
	void draw() { cout << "circle"; }
	void rotate(double degree) {}
};

template<typename C, typename Oper>
void for_all(C& c, Oper op)
{
	for (auto& x : c)
		op(x);
}
```

Function Object - Usage

```
vector<unique_ptr<Shape>> v;
v.push_back(make_unique<Rect>());
v.push_back(make_unique<Circle>());

for_all(v, [](unique_ptr<Shape>& s){ s->draw(); });
for_all(v, [](unique_ptr<Shape>& s){ s->rotate(45); });
```

Variadic Templates

```
void foo() {
      cout << endl;
}

template<typename T>
void bar(T x) {
      cout << x << " ";
}

template <typename T, typename ...Tail>
void foo(T head, Tail... tail) {
      bar(head);
      foo(tail...);
}
```

```
foo(1, 2, 2, "Hello");
foo(0.2, 'c', "yuck!", 0, 1, 2);
```

Function Specialization

```
template <class T>
T MyMax(T a, T b)
{
      return a > b ? a : b;
}

template <>
const char* MyMax(const char* a, const char* b)
{
      return strlen(a) > strlen(b) ? a : b;
}
```

```
assert(MyMax(1, 2) == 2);
assert(MyMax("Morning", "Afternoon") == "Afternoon");
```

Function Specialization Example

```
struct IO {
	int addr;
	bool state;
};

struct Servo {
	int axis;
	double position;
};

struct Verify {
	template<typename T>
	static void Equal( T arg ) { throw exception("Should specialize"); }

	static void Equal(IO io) {
		cout << "check addr " << io.addr << " to be " << io.state << endl;
	}

	static void Equal(Servo s) {
		cout << "check axis " << s.axis << " position is at " << s.position << endl;
	}
};
```

```
Verify::Equal(IO{ 41, true });
Verify::Equal(Servo{ 3, 3.141592 });
```

Template Class Specialization CompileTimeAssert

```
template<bool> struct CompileTimeAssert;
template<> struct CompileTimeAssert<true> {};
```

```
// CompileTimeAssert< sizeof(uint32_t) == 2  >();
CompileTimeAssert< sizeof(uint32_t) == 4 >();

// CompileTimeAssert< std::is_base_of<Rect, Shape>::value >();
CompileTimeAssert< std::is_base_of<Shape, Rect>::value >();

static_assert(std::is_base_of<Shape, Rect>::value, “Rect is a Shape");
```

Template Class Specialization TypeTrait

```
template< typename T >
struct IsVoid {
static const bool value = false;
};

template<>
struct IsVoid< void >{
static const bool value = true;
};

template< typename T >
struct IsPointer{
static const bool value = false;
};

template< typename T >
struct IsPointer< T* >{
static const bool value = true;
};
```

Template Class Specialization TypeTrait - usage

```
CompileTimeAssert< IsVoid<void>::value >();
// CompileTimeAssert< IsVoid<int>::value >();

CompileTimeAssert< IsPointer<Shape*>::value >();
// CompileTimeAssert< IsPointer<Shape>::value >();


static_assert(is_pointer<Shape*>::value, "Shape* should be a pointer");
```