항목 3-10

function object

1. 기본 개념
2. 함수 객체의 장점
3. 상태를 가지는 함수
4. 각각의 함수 객체는 자신만의 타입을 가진다.
5. 특정 상황에서 함수 객체는 함수 보다 빠르다.
6. 단위 전략 설계와 함수 객체
7. STL 과 함수 객체
8. 익명의 함수 객체 "람다"

1. 기본 개념

C++ 에서는 함수 호출시에 사용하는 ( )도 결국 연산자 이다. 따라서 클래스를 만들 때 ( )연산자를 재정의 하는 것도 가능하다. ( )연산자를 재정의 하면 객체를 마치 함수 처럼 사용할 수 있는데 이를 **함수 객체 (function object, functor)** 라고 부른다.

"함수객체"의 정확한 의미는 ()연산자를 사용해서 함수 처럼 호출가능한 모든 객체들을 이야기 한다. 즉, 함수 포인터, 함수에 대한 참조, 멤버 함수 포인터 등도 함수객체에 포함된다. 하지만 본 교재 에서는 ()연산자를 재정의한 클래스(구조체)로 제한해서 사용하였다.

아래 예제는 임의의 타입에 대해 값 2개를 인자로 받아서 합을 구하는 함수 객체 이다.

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

template<typename T> struct plus
{
    T operator()( const T& lhs, const T& rhs ) const
    {
        return lhs + rhs;
    }
};
int main()
{
    plus<int> p;
    int n1 = p(1,2);// p는 객체지만 함수처럼 사용한다.
                    // p.operator()(1,2) 처럼 해석 된다.
    int n2 = p.operator ()(1,2); // 이렇게 사용 해도 된다.

    cout << n1 << " " << n2 << endl;
}
```

위 코드에서 p는 함수가 아닌 plus<int> 타입의 객체 이다. 하지만 ()연산자를 재정의 하고 있어서 마치 함수 처럼 사용 할 수 있다. 신기해 보이지만 원리는 지극히 단순하다. 결국 아래의 표현은

```
int n1 = p(1,2);
```

아래 처럼 해석된다.

```
int n1 = p.operator ()(1,2);
```

2. 함수 객체의 장점

STL에 사용하다 보면 이런 함수객체를 자주 사용하게 된다. 그럼, 왜 함수를 사용하지 않고 함수객체를 사용할까 ?

함수를 사용하는 것보다 함수 객체를 사용하면 아래와 같은 장점이 있다.

- **상태를 가지는 함수를 만들 수 있다.**
- **각각의 함수객체는 자신만이 타입을 가진다.**
- **특정상황에서 함수객체는 함수 보다 빠르다.**

각각의 장점에 대해서 자세히 살펴 보자.

3. 상태를 가지는 함수

간단한 예제를 하나 생각해 보자. 3개의 서로 다른 임의의 난수를 구한다고 생각해 보자. rand() 함수를 사용하면 된다. 하지만 이 함수는 중복된 숫자가 나올 수도 있으므로 항상 중복을 조사해야만 한다.

호출 할 때 마다 계속 서로 다른 난수가 나오는(즉, 이전에 구한 난수와 중복되지 않은) 함수가 있다면 편리하지 않을까 ? 매번 새로운 난수가 나오려면 이전에 나왔던 모든 값을 어딘 가에 기록해 두어야 한다. 즉 자신의 상태를 기억하고 있어야 한다.

아래 URandom<> 함수 객체를 생각해 보자.

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

template<int N> class URandom
{
public:
    URandom(bool b = false) : recycle(b)
    {
        srand(time(0));
        random_map.set();       // bitset의모든비트를1로초기화한다
    }

    int operator()()
    {
        if ( random_map.none() )//
        {
            if ( ! recycle )
                return -1;      // 더이상새로운난수가없음

            random_map.set();
        }
        int n = -1;

        while ( 1 )
        {
            n = rand() % N;
            if ( ! random_map.test(n) )
                continue;
            random_map.reset(n);
```

```
            break;
        }
        return n;
    }
private:
    bitset<N> random_map;
    bool  recycle;
};

int main()
{
    URandom<10> r1; // 0~9 사이의 중복을 허용하지 않는 난수를 구하는 함수객체
                    // 10개의 모든 난수를 사용하면 -1리턴.
    for ( int i = 0; i < 15; ++i)
        cout << r1() << " ";

    cout << endl;

    URandom<10> r2(true);

    for ( int i = 0; i < 15; ++i)
        cout << r2() << " ";

    cout << endl;
}
```

URandom<> 함수 객체는 STL의 bitset<>을 사용해서 현재 까지 사용한 난수를 기록하고 있다가 다음번에 난수를 구할 때 사용하고 있다. URandom<>는 결국 객체 이므로 멤버 data를 가질수가 있기 때문에 이런 기법이 가능하다. 일반 함수를 사용해서 이렇게 동작하도록 만들수 있을까 ?

일반 함수는 동작을 표현 할 수는 있지만 상태를 가질 수는 없다. 하지만, 함수 객체는 결국 객체 **이므로 상태(멤버 데이터)를 가질수 있고** 생성자, 소멸자, 복사 생성자, 대입 연산자등을 사용할 수 있으므로 함수 객체의 생성, 소멸, 대입 등의 과정을 제어할 수도 있다.

4. 각각의 함수객체는 자신 만의 타입을 가진다.

일반 함수는 signature(리턴 타입과 파라미터 list)가 동일 하면 모두 같은 타입이다. 따라서 오버로딩이나 템플릿 인자 등으로 사용하기에는 적합하지 않다.
도대체 무슨 말일까? 아래 코드를 생각해 보자.

```
// foo, goo는 분명 다른 함수이지만
// 함수 signature가 같기 때문에 동일 타입이다.
void foo() { }
void goo() { }

void test( void(*)(void) )
{
    cout << "test" << endl;
}

void main()
{
    // foo, goo는다른함수지만결국같은타입이다. void(*)(void)
    // 아래의A, B가서로다른함수를호출하게만들수는없다.
```

```
    test( foo ); // A
    test( goo ); // B
}
```

위의 예제에서 foo, goo는 분명 다른 함수이지만 함수 모양(signature)가 동일 하므로 같은 타입이 된다. 따라서 A, B는 모두 동일한 test 함수를 호출하게 된다.

이번에는 함수 객체의 경우를 생각해 보자.

```
struct FOO
{
    void operator()() { }
};
struct GOO
{
    void operator()() { }
};
// FOO, GOO는 서로 다른 타입이므로 test 함수의 인자로 사용해서 오버로딩을 할 수 있다.
void test( FOO )
{
    cout << "f2(FOO)" << endl;
}
void test( GOO )
{
    cout << "f2(GOO)" << endl;
}
int main()
{
    FOO foo;
    GOO goo;

    f2( foo ); // A
    f2( goo ); // B
}
```

FOO, GOO는 모두 void를 인자로 가지고 void를 리턴하는 함수 객체 이다. 이 경우 분명 함수 signature는 동일 하지만 FOO, GOO는 다른 타입이다. 따라서 A, B는 서로 다른 test 함수를 호출하게 된다.

FOO, GOO 가 서로 다른 타입이므로 test함수를 오버로딩으로 만들 수 있게 된다.
그럼, 이렇게 오버로딩을 해서 얻게 되는 장점은 도대체 무엇일까 ? 다음 단계로 가보자.

5. 특정 상황에서 함수객체는 함수 보다 빠르다.

"함수객체는 함수 보다 빠르다." 정말 중요한 말인데, 좀 어렵기 때문에 많은 사람들이 잘 모르고 있다.
지금부터 이 말의 의미를 정확히 살펴보자. 아래 예제를 보자.

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

inline int plus( int a, int b)
{
    return a + b;
}
inline int minus( int a, int b)
```

```
{
    return a - b;
}
// 컴파일 시간에 인자로 plus가 전달될지 minus가 전달 될지 알 수 있을까?
void test( int(*f)(int, int) )
{
    int n = f(2,1);          // (A) inline 치환이 될 수 있을까?
    cout << n << endl;
}

int main()
{
    int n1 = plus( 2,1);     // (B) inline 치환이 된다.

    int k;
    cin >> k;

    if ( k == 1 )
        test( plus);
    else
        test( minus );
}
```

plus(), minus() 함수의 경우는 template으로 구현하는 것이 좋지만, 현재 주제에 집중하기 위해서 단순하게 int 인자를 사용하였다.

plus(), minus() 함수는 구현이 간단하므로 inline함수로 구현하였다. 따라서 (B)는 당연히 호출이 아니라 인라인 치환이 발생한다. 그런데, (A)는 어떻게 될까? 인라인 치환은 실행시간이 아닌 컴파일 시간에 적용되는 문법이다.(항목 1-11 참고) 그런데, test() 함수에 plus가 전달될지 minus가 전달될 지 컴파일 시간에 알 수 있을까?

결국 일반 함수를 다른 함수의 인자로(함수포인터를 사용해서) 전달해서 사용할 때 인라인 치환은 될 수가 없다.(항목 1-11 참고)

함수 객체의 경우는 어떨까 ?

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

struct plus
{
    inline int operator()(int a, int b) const
    {
        return a + b;
    }
};
struct minus
{
    inline int operator()(int a, int b) const
    {
        return a - b;
    }
};
// plus, minus는 서로 다른 타입이다. 따라서 test()함수도 2개 필요하다.
// plus를인자로가지는test
void test( plus f )
{
```

```
    int n = f(2,1); // 컴파일 시간에 f가 무슨 타입인지 컴파일러는 정확히 알 수 있다.
                    // 따라서 inline 치환에는 전혀 문제가 없다.
    cout << n << endl;
}
// minus를인자로가지는test
void test( minus f )
{
    int n = f(2,1);
    cout << n << endl;
}

int main()
{
    plus p;
    minus m;

    test( p );
    test( m );
}
```

이 경우는 test(plus), test(minus) 함수를 컴파일 할 때 컴파일러는 인자의 타입을 정확히 알 수 있으므로 인라인 치환을 할 수 있게 된다. 또한 2개의 test() 함수는 하는 일이 완전히 동일 하므로 template을 사용해서 만드는 것이 편하다.

```
template<typename T> void test( T f )
{
    int n = f(2,1);
    cout << n << endl;
}
```

결국, 임의의 함수를 다른 함수에 인자로 전달 한 후 사용할 때 일반함수는 인라인화 될 수 없지만 함수 객체는 인라인화 될 수 있다.

그럼, 인라인 치환이 왜 중요 할까 ? 다음 단계에서 살펴 보자.

6. 단위 전략 설계(Policy Base Design)와 함수객체

다른 많은 개발자들이 사용하게 될 sort() 함수를 만든다고 가정해 보자. 가장 중요한 점은 무엇일까 ?

1. 속도가 빨라야 한다. 라이브러리에 있는 함수가 사용자가 직접 만든 함수보다 현격히 성능이 떨어 진다면 아무도 사용하지 않을 것 이다.
2. 범용적으로 사용할 수 있어야 한다. 사용자는 내림차순의 sort가 필요 한데, 라이브러리의 함수가 오름차순 만을 제공한다면 문제가 될 것이다.

먼저 아래의 sort를 생각해 보자.

물론 quick sort 알고리즘이 가장 빠르겠지만 "함수객체"라는 지금의 주제에 충실하기 위해 되도록 간단한 알고리즘을 사용했다.

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

void Sort( int* s, int n )
```

```
{
    for ( int i = 0; i < n-1 ; ++i )
        for ( int j = i + 1; j < n; ++j )
            if ( s[i] < s[j] )
                swap( s[i], s[j]);
}

int main()
{
    const int size = 10;
    int x[size] = { 1,3,5,-7,9,-2,4,-6,8,10};

    Sort( x, size );

    for ( int i = 0; i < size; ++i )
        cout << x[i] << " ";

    cout << endl;
}
```

그런데, 이 구현에는 약간 문제가 있다. Sort()의 구현이 내림차순으로 동작되도록 만들어 져 있기 때문에 오름차순의 Sort()가 필요할 때는 사용할 수가 없다. 가장 좋은 방법은 Sort() 를 수행할 때 2개의 요소를 어떻게 비교할 것인지를 지정할 수 있으면 좋을 거 같다. 2개의 요소를 비교하는 함수를 Sort()의 3번째 인자로 전달해 보자. 수정된 Sort()이다.

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

void Sort( int* s, int n, bool(*cmp)(int, int) )
{
    for ( int i = 0; i < n-1 ; ++i )
        for ( int j = i + 1; j < n; ++j )
            if ( cmp(s[i], s[j]) )
                swap( s[i], s[j]);
}

// 2개의 요소를 비교하는 전략을 담은 일반 함수
bool less ( int a, int b )
{
    return a < b;
}

int main()
{
    const int size = 10;
    int x[size] = { 1,3,5,-7,9,-2,4,-6,8,10};

    Sort( x, size, less);

    for ( int i = 0; i < size; ++i )
        cout << x[i] << " ";

    cout << endl;
```

```
}
```

자, 이제 Sort()를 수행할 때 내림차순으로 할지 오름차순으로 할지 또는 다른 기준으로 할지를 Sort()의 사용자가 결정할 수 있게 되었다. 보다 범용적으로 사용할 수 있게 되었다.

그런데, 여기에는 또 다른 문제가 있다. 첫 번째 버전의 Sort()는 2개의 요소를 비교하기 위해 "<" 연산자를 사용했는데 2번째 버전에서는 cmp(s[i], s[j]) 로 함수를 호출하게 된다. 당연히 함수 호출에 따른 속도 저하가 발생한다. 수백만 건을 Sort()해야 한다면 엄청난 속도의 저하가 발생 하게 된다. 즉, 함수 포인터를 사용해서 비교 함수를 인자로 받으므로서 **범용성을 얻었지만 속도 저하가 문제가 생기게 된다**.

속도를 빠르게 하기 위해서 비교 함수를 인라인으로 만들자고 생각할 수도 있지만 이전 항목에서 보았듯이 인자로 전달한 후 호출하면 인라인 치환이 될 수 없다.

이제, 해결책은 한가지 밖에 없다. Sort()에 전달할 함수를 일반함수가 아닌 함수 객체로 만들면 된다. 아래는 최종적으로 완성한 Sort()이다. 물론 서로 다른 함수객체를 인자로 가지기 위해서는 Sort()는 template 으로 만들어야 한다.

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

// 이제 비교 함수를 함수객체로 만듭니다.
template<typename T> struct less
{
    inline bool operator()(const T& lhs, const T& rhs) const
    {
        return lhs < rhs;
    }
};

template<typename CMP> void Sort( int* s, int n, CMP cmp )
{
    for ( int i = 0; i < n-1 ; ++i )
        for ( int j = i + 1; j < n; ++j )
            if ( cmp(s[i], s[j])  )
                swap( s[i], s[j]);
}

int main()
{
    const int size = 10;
    int x[size] = { 1,3,5,-7,9,-2,4,-6,8,10};

    less<int> cmp;
    Sort( x, size, cmp ); // 함수객체를 인자로 전달합니다

    for ( int i = 0; i < size; ++i )
        cout << x[i] << " ";

    cout << endl;
}
```

이제 위 Sort() 함수는 사용자가 직접 만든 것 만큼(Sort 함수 안에서 '<' 연산자를 직접 사용한 것만큼) 빠르게 동작한다.

7. STL과 함수 객체

STL의 <functional> 헤더 안에는 미리 만들어진 많은 함수 객체가 있다. 또한 <algorithm> 헤더 파일에는 위처럼 만들어진 sort()함수(quick sort알고리즘)가 있다. 따라서, 아래 처럼 사용할 수 있다.

```
#include <algorithm> // sort() 함수가 제공된다.
#include <iostream>
#include <functional> // 다양한 함수 객체가 제공된다.
using namespace std;

int main()
{
    int x[10] = { 1,-2,3,-4,5,-6,7,-8,9,10 };

    greater<int> g;     // '>' 연산을 수행하는 함수객체

    sort( x, x+10, g );

    for ( int i = 0; i < 10; ++i )
        cout << x[i] << " ";

    cout << endl;
}
```

STL이 제공하는 sort() 함수는 2번째 인자로 요소의 개수 대신에 마지막 다음 요소의 주소가 인자로 전달된다. 자세한 이야기는 8장 STL에서 다룬다.

C 표준 함수인 qsort()가 일반함수를 사용하지만 STL의 sort() 함수는 함수객체를 사용하기때문에 인라인 치환의 효과를 볼 수 있다. 따라서, STL의 sort() 함수가 C표준의 qsort()함수 보다 빠르다.

STL의 다른 함수객체에 대한 자세한 이야기는 8장에서 자세히 살펴보자.

8. 익명의 함수 객체 – "람다"

Sort()함수에 비교인자를 지정하기 위해서 매번 함수 객체를 만들어야 하는 것은 좀 불편하다. 물론 STL안에 자주 사용하는 함수객체를 라이브러리 형태로 제공하지만 사용자가 원하는 형태가 없다면 만들어야 한다.

차세대 C++ 표준인 C++0x에는 한번 사용할 간단한 함수객체를 쉽게 만들 수 있는 "람다"라는 개념을 도입했다. 람다를 사용하면 아주 쉽게 Sort()인자에 전달할 함수객체를 만들 수 있다.
아래 예제는 주어진 배열을 절대값순서로 sort()하는 예제이다. 비교 전략으로 사용할 함수 객체를 "람다"를 사용해서 만들었다.

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

int main()
{
    int x[10] = { 1,-2,3,-4,5,-6,7,-8,9,10 };

    sort( x, x+10, [](int x, int y) { return abs(x) < abs(y);} ); // 람다는 []로 시작한다.

    for ( int i = 0; i < 10; ++i )
        cout << x[i] << " ";
```

```
    cout << endl;
}
```

위 예제는 "람다"에 대한 간단한 소개 정도이다. "람다"의 자세한 문법은 항목 12-03을 참고 하자.

핵심 정리

- **함수객체는 상태를 가지는 함수를 만들 수 있고, 일반 함수 보다 속도도 빠르다.**
- **함수객체를 사용하면 성능의 저하 없이 정책을 인자화 해서 넘길 수 있다.**