Advanced C++

교재 Sample

㈜ 아임구루 부설 아이오 교육센터 (www.ioacademy.co.kr)

항목 1 - 03

객체의 복사 기술

핵심개념

- 객체의 얕은 복사(Shallow Copy) 현상을 이해하고 4 가지 주된 해결책을 이해 한다.
- 깊은 복사 (Deep Copy)
- 참조 계수 (Reference Counting)
- 소유권 이전의 복사 전략을 이해하고 C++11 Move Semantics 의 개념을 이해 한다.
- 복사 금지의 정책의 특징을 이해하고 C++11 delete function 문법의 탄생 배경을 이해 한다.
- C++ 표준 라이브러리가 제공하는 다양한 클래스의 복사 정책을 이해 한다.
- 참조 계수 정책 에서 Copy On Write(COW) 기술을 이해 한다.

참고자료

- Effective C++ 2/E 항목 14

1. 디폴트 복사 생성자 와 객체의 얕은 복사(Shallow Copy) 현상

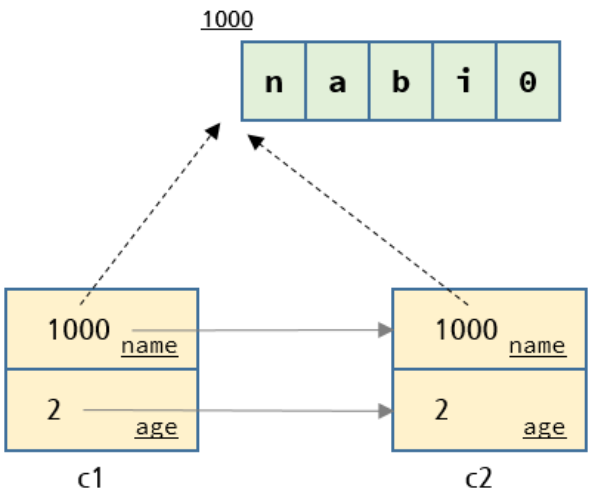
C++ 컴파일러는 사용자가 복사 생성자를 제공하지 않으면 디폴트 복사 생성자를 제공.

모든 멤버를 복사(bitwise copy).

포인터를 멤버 data 로 가지는 클래스의 경우에는 실행시간 문제가 발생할 수 있다

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

class Cat
{
	char* name;
	int   age;
public:
	Cat(const char* s, int a) : age(a)
	{
		name = new char[strlen(s) + 1];
		strcpy(name, s);
	}
	~Cat() { delete[] name; }
};
int main()
{
	Cat c1("nabi", 2);
	Cat c2 = c1;  // 얕은 복사(Shallow Copy)
		      // 컴파일에는 문제가 없지만 실행하면 에러 발생
}
```



디폴트 생성자가 c1 의 모든 멤버를 c2 를 복사하는 경우는 메모리 그림

c2 의 소멸자에서 delete[] name 으로 이름을 담은 메모리를 삭제 하므로 더 이상 c1 을 사용할 수 없다

c1 이 파괴 될 때 delete[] name 을 한번 더하게 되므로 동일 메모리를 2 번 삭제 하게 된다.

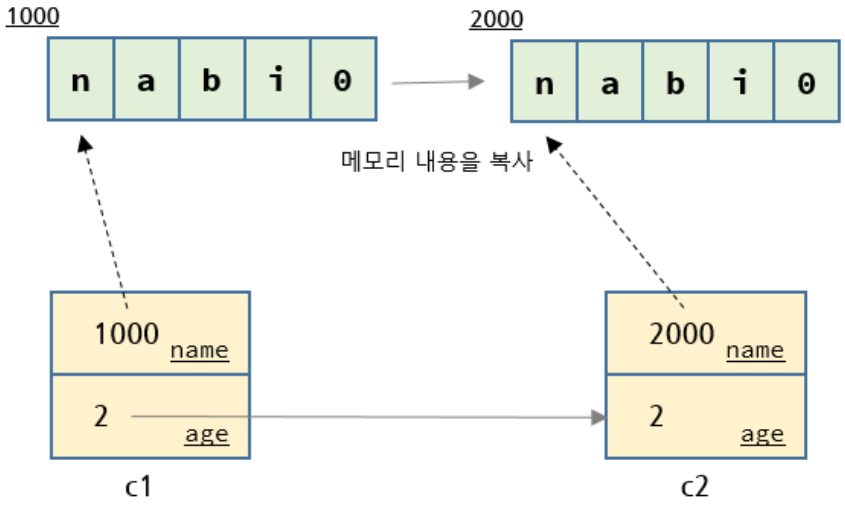
반드시 알아 두어야 할 점

- 클래스 안에 포인터 멤버가 있는 경우 얕은 복사(Shallow Copy) 현상이 있을 수 있다. 복사 생성자를 제공해서 해결해야 한다.
- 깊은 복사, 참조 계수, 소유권이전(move), 복사 금지의 기술이 있다.

2. 깊은 복사(Deep Copy)

포인터가 아닌 멤버는 값을 복사하고 포인터 멤버는 새로운 메모리를 할당하고 메모리 자체를 복사하는 개념

```
Cat(const Cat& c) : age(c.age)
{
	name = new char[strlen(c.name) + 1];
	strcpy(name, c.name);
}
```



깊은 복사를 사용하는 경우는 메모리 그림

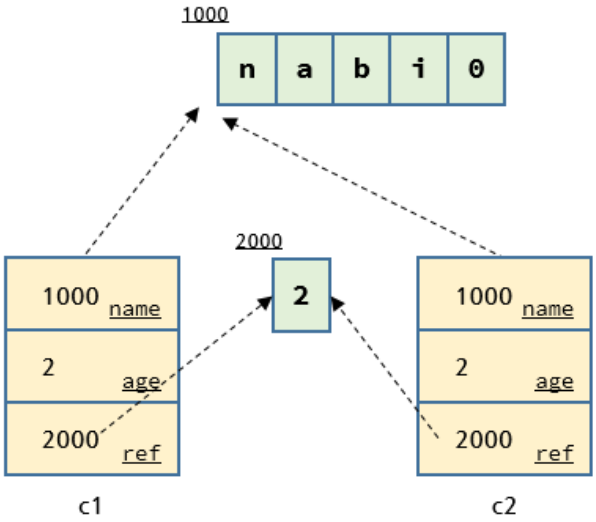
C1, c2 는 파괴될 때 각각 자신의 자원(메모리)를 해지 하므로 아무 문제 없다.

하지만, 동일한 자원을 메모리에 여러 번 생성 하게 되는 단점이 있다.

3. 참조 계수 (Reference Counting)

동일한 자원은 메모리에 한번 만 만든 후에 모든 객체가 공유하고 대신 동일 자원을 몇 개의 객체가 사용하는지 개수를 관리하는 기법.

```
class Cat
{
	char* name;
	int   age;
	int*  ref;
public:
	Cat(const Cat& c) : name(c.name), age(c.age), ref(c.ref) { ++(*ref); }
	~Cat()
	{
		if (--(*ref) == 0)
		{
			delete[] name;
			delete ref;
		}
	}
	Cat(const char* s, int a) : age(a)
	{
		name = new char[strlen(s) + 1];
		strcpy(name, s);
		ref = new int(1);
	}
};
int main()
{
	Cat c1("nabi", 2);
	Cat c2 = c1;
}
```



참조 계수로 자원을 관리할 때 c1 이 자신의 이름을 변경하면 어떻게 될까 ?

c1 이 자원(name)을 변경하기 전에 자원의 복사본을 만든 후에 변경해야 한다. 즉 자원을 변경(Write) 하기전에 복사본(Copy)를 만들어야 한다.

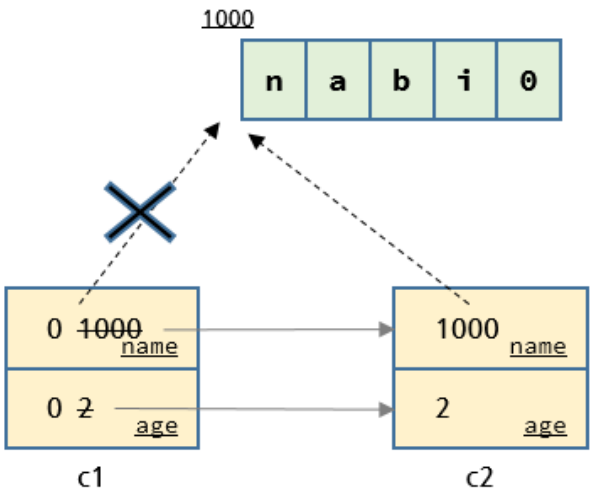
COW(Copy On Write) 개념 – 4 장 간접층의 원리 참고

4. 소유권 이전 (Move Semantics)

모든 멤버를 얕은 복사 후에 원본 객체의 멤버를 0으로 변경한다.

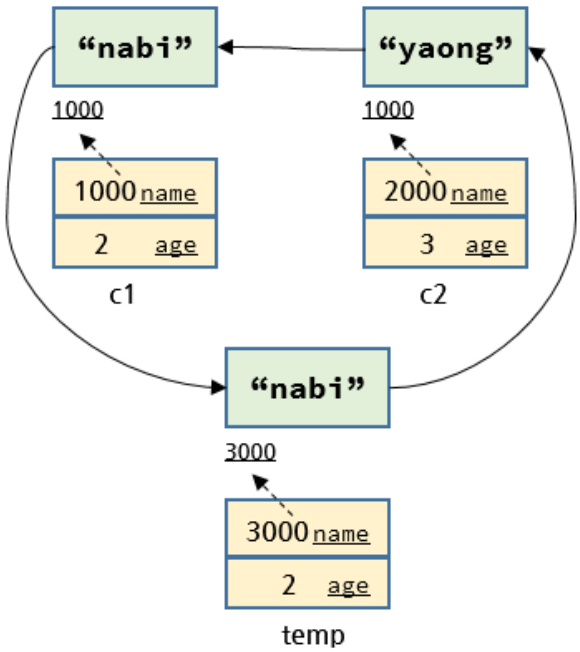
자원을 복사하지 않고 전달(move) 하는 개념

```
class Cat
{
	char* name;
	int   age;
public:
	// 소유권 이전의 복사 생성자
	Cat(Cat& c)    : name(c.name), age(c.age)
	{
		c.name = 0;
		c.age = 0;
	}
	Cat(const char* s, int a) : age(a)
	{
		name = new char[strlen(s) + 1];
		strcpy(name, s);
	}
	~Cat() { delete[] name; }
};
int main()
{
	Cat c1("nabi", 2);
	Cat c2 = c1;  // c1의 자원은 c2로 이동된다. c1은 더 이상 사용할 수 없다.
}
```

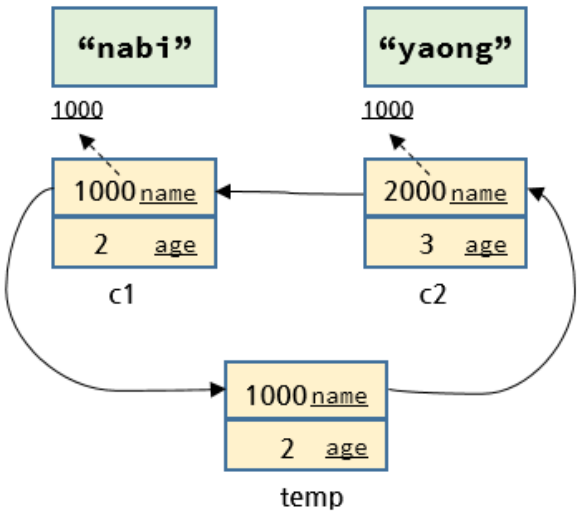


c1 이 사용하든 모든 자원은 c2 에 전달하고 c1 은 0 으로 변경한다.

swap 등의 일부 알고리즘은 복사(copy) 보다는 이동(move) 할 때 성능이 더 좋다.



복사(Copy)를 사용한 swap 구현



이동(Move)를 사용한 swap 구현

하지만 항상 move 가 필요 하지는 않고 복사가 필요 할 때 가 있다.

C++11 은 복사 생성자와 Move 생성자 모두를 제공할 수 있다. rvalue reference 개념의 등장

```
class Cat
{
	char* name;
```

```
        int   age;
public:
        // 생성자(Constructor)
        Cat(const char* s, int a) : age(a)
        {
                name = new char[strlen(s) + 1];
                strcpy(name, s);
        }
        // 복사 생성자(Copy Constructor)
        Cat(const Cat& c)
        {
                // 참조계수 또는 깊은 복사 사용해서 구현...
        }
        // 이동 생성자(Move Constructor)
        Cat(Cat&& c) : name(c.name), age(c.age)
        {
                c.name = 0;
                c.age = 0;
        }
        ~Cat() { delete[] name; }
};
int main()
{
        Cat c1("nabi", 2);
        Cat c2 = c1;              // 복사 생성자 호출
        Cat c3 = std::move(c1); // move 생성자 호출
}
```

더 자세한 이야기는 5 장의 rvalue reference 와 move semantics 참고.

5. 복사 금지 정책과 delete function

싱글톤, 스마트포인터, mutex 등 일부 클래스의 객체는 복사될 필요가 없다. 복사를 막아야 한다.

복사 생성자를 private 영역에 선언만 한다. 구현을 제공하지 않는다.

외부에서 복사 하려고 하면 private 함수에 접근할 수 없으므로 컴파일 에러 발생.

자신의 멤버 함수에서 복사 하려고 하면 복사 생성자의 구현이 없으므로 링크 에러.

외부와 내부 모두에서 복사 할 수 없다.

```
class Cat
```

```
{
	char* name;
	int   age;

	// 복사와 대입 금지의 기술
	Cat(const Cat& c);
	void operator=(const Cat& c);
public:
	void foo()
	{
		Cat c1("nabi", 2);
		Cat c2 = c1;  // private 영역에 접근할수 있지만. 구현이 없으므로 링크 에러
	}
	Cat(const char* s, int a) : age(a)
	{
		name = new char[strlen(s) + 1];
		strcpy(name, s);
	}
	~Cat() { delete[] name; }
};
int main()
{
	Cat c1("nabi", 2);
	Cat c2 = c1;		// 컴파일 에러..
}
```

복사금지의 재사용 기술

상속을 사용한 복사 금지 - boost 라이브러리. uncopyable.hpp

```
class noncopyable
{
protected:
	noncopyable() {}
	~noncopyable() {}
private:
	// emphasize the following members are private
	noncopyable(const noncopyable&);
	const noncopyable& operator=(const noncopyable&);
};
class Cat : public noncopyable {};

int main()
{
	Cat c1("nabi", 2);
	Cat c2 = c1; // error
}
```

매크로 사용한 복사 금지 - webkit 등의 오픈소스

```
#ifndef WTF_Noncopyable_h
#define WTF_Noncopyable_h

#include <wtf/Compiler.h>

#if COMPILER_SUPPORTS(CXX_DELETED_FUNCTIONS)
#define WTF_MAKE_NONCOPYABLE(ClassName) \
       private: \
           ClassName(const ClassName&) = delete; \
           ClassName& operator=(const ClassName&) = delete;
#else
#define WTF_MAKE_NONCOPYABLE(ClassName) \
       private: \
           ClassName(const ClassName&); \
           ClassName& operator=(const ClassName&)
#endif

#endif // WTF_Noncopyable_h
```

C++11 의 delete function 과 default function 문법

복사(대입) 금지의 테크닉을 문법화 한다.

```
class Point
{
       int x, y;
public:
       Point(int a, int b) : x(a), y(b) {}

       Point() = default;
       Point(const Point& p) = delete;
       Point operator=(const Point& p) = delete;

       Point(Point&& p) = default;
       Point& operator=(Point&& p) = default;
};

int main()
{
       Point p1(1, 2); // ok. 사용자가 제공
       Point p2;       // ok. default function 문법

       Point p3( p1 );  // error. 복사생성자 제거, delete function
```

```
        Point p4(move(p1)); // ok.    기본 move 생성자 제공

        p2 = p1;       // error. 복사 대입연산자 제거
        p4 = move(p2);// ok.    이동 대입연산자 제공
}
```

더 자세한 이야기는 5장 C++11 delete function 참고

6. 표준 라이브러리 클래스와 복사 정책

	얕은 복사	깊은 복사	참조 계수	이동(move)	복사 금지
complex<>	O	X	X	X	X
shared_ptr<>	X	X	O	X	X
unique_ptr<>	X	X	X	O	O
mutex	X	X	X	X	O
vector<> list<>	X	O	X	O	X

```
int main()
{
        vector<int> v1(10, 3);
        vector<int> v2 = v1;      // 깊은 복사
        vector<int> v3 = move(v1); // 이동
}
```

항목 3 - 08

Generic 프로그램과 STL 의 설계 철학

핵심개념

- 일반화 (Generic) 프로그램의 개념을 이해하고 STL 의 설계 철학을 이해 할 수 있다.
- STL 의 find() 알고리즘의 설계 방식을 이해하고 일반화 함수를 만들 수 있다.
- 컨테이너와 일반화 알고리즘을 연결하는 반복자를 이해하고 구현할 수 있다.
- Iterator category 와 iterator tratis 개념을 이해 한다.

참고자료

- Generic Programming And STL , Matthew H Austern

1. STL 의 find 함수 만들기

Step 1. C 언어의 strchr() 함수

문자열에서 특정 문자를 검색하는 함수이다.

문자 검색에 성공할 주소를 리턴하고 실패할경우 NULL 을 리턴한다. NULL 문자도 검색 가능하다.

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

char* xstrchr(char* s, char c)
{
	while (*s != c && *s != 0)
		++s;

	return *s != c ? NULL : s;
}

int main()
{
	char s[] = "abcdefg";
	char* p = xstrchr(s, 'c');
}
```

이 함수는 2 개의 문제점이 있다.

검색에 포함되는 모든 문자는 NULL 문자로 종료 되어야 한다.

전체 문자열 검색만 가능하다. 부분 문자열로부터 검색할 수가 없다.

Step 2. 검색 구간의 일반화 - 부분 문자열 검색이 가능하게.

검색의 시작 뿐 아니라 검색의 끝도 전달한다.

```
char* xstrchr(char* first, char* last, char value)
{
	while (*first != value && first != last)
		++first;

	return *first != value ? NULL : first;
}
int main()
{
```

```
        char s[] = "abcdefg";
        char* p = xstrchr(s, s+2, 'c'); // "abc"에서 검색

        if (p == NULL) cout << "찾을수 없습니다." << endl;
        else           cout << *p << endl;
}
```

장점 : 전체 문자열과 부분 문자열 모두에서 검색 가능하다.

단점 : 문자열 검색만 가능하다. 다른 타입의 배열은 검색할 수 없다.

Step 3. 검색 대상 타입의 일반화 - 모든 타입의 배열을 검색할 수 있도록

함수 템플릿을 사용한다.

```
template<typename T> T* xstrchr(T* first, T* last, T value)
{
        while (*first != value && first != last)
                ++first;

        return *first != value ? NULL : first;
}

int main()
{
        double d[] = { 1,2,3,4,5,6,7,8,9,10 };
        double* p = xstrchr(d, d + 4, 5.0); // { 1,2,3,4,5 } 에서 검색
        cout << *p << endl;
}
```

장점 : 문자열 뿐 아니라 모든 타입의 임의의 구간에서 검색이 가능 하다.

단점 : 구간의 타입과 찾고자 하는 값의 타입이 연관되어 있다. (double 배열에서 int 를 찾을수 없다.)

템플릿 인자로 T* 라고 표시하면 진짜 포인터만 사용해야 한다.(스마트 포인터 사용불가)

모든 타입의 배열에서 선형검색을 수행하므로 함수 이름이 적합하지 않다.

```
double* p1 = xstrchr(d, d + 4, 5); // xstrchr(T*, T*, T) 가 되어야 하는데
                                   // double*, double*, int 이므로 컴파일 시간 에러
```

```
smart_pointer<double> first(d);
smart_pointer<double> last(d+4);
double* p2 = xstrchr(first, last, 5.0);//first,last는 포인터가 아니므로 error
```

Step 4. 보다 일반화 되게!

진짜 포인터 뿐 아니라 스마트 포인터도 사용 가능하게 한다.

구간의 타입과 찾고자 하는 요소의 타입을 분리해서 double 배열에서 int를 찾을수 있게 한다.

함수 이름을 xfind()로 변경한다.

```
template<typename T1, typename T2> T1 xfind(T1 first, T1 last, T2 value)
{
	while (*first != value && first != last)
		++first;

	return *first != value ? NULL : first;
}
int main()
{
	double d[] = { 1,2,3,4,5,6,7,8,9,10 };
	double* p = xfind(d, d + 4, 5); // 5는 int 이지만 검색이 가능하다.
	cout << *p << endl;
}
```

장점 : 진짜 포인터 뿐 아니라 스마트 포인터 등의 객체도 사용 가능하다.

구간의 타입과 찾고자 하는 요소의 타입은 연관되지 않았다. double 배열에서 int를 찾을 수 있다.

단점 : 실패시 NULL 포인터(0)을 리턴 하고 있다. 구간은 포인터 뿐 아니라 객체(스마트포인터 등)로 표현될 수 있다. 검색 실패를 알려주는 다른 방법을 사용하자.

Step 5. 검색 실패시 NULL 대신 last 리턴

first ~ last의 검색 구간 중 last를 검색에 포함 시키지 않는다.

검색에 성공 했다면 last가 나오지 않는다. 검색 실패 시 last를 리턴 한다.

```
template<typename T1, typename T2> T1 xfind(T1 first, T1 last, T2 value)
{
	while ( first != last && *first != value)
```

```
            ++first;
        return first;
}
int main()
{
        double d[] = { 1,2,3,4,5,6,7,8,9,10 };

        double* p = xfind(d, d + 9, 10); // {1,2,3,4,5,6,7,8,9,10} 이지만
                                         // 10은 검색에 포함되지 않는다.
                                         // 전체 검색이 필요하면 x ~ x+10 까지 검색해야 한다.
        if (p == d + 9) cout << "찾을 수 없습니다." << endl;
        else            cout << *p << endl;
}
```

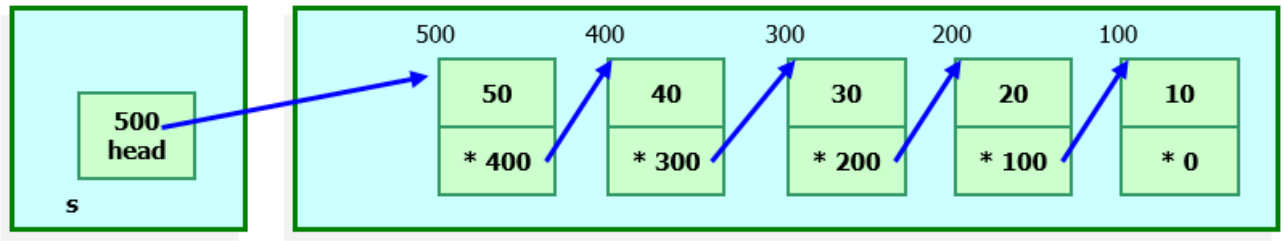
xfind() 함수 총정리

- 모든 타입의 배열에서 [first, last) 구간에서 선형 검색을 수행 한다.
- last 는 검색 대상은 아니다. last 를 past the end 라고 한다
- 구간의 표현은 진짜 포인터 뿐 아니라 ++, !=, ==, * 연산자가 재정의된 객체도 사용 가능하다.
- 검색에 성공할 경우 위치를 검색에 실패할 경우 last 를 리턴 한다.
- 구간의 타입과 찾고자 하는 요소의 타입은 반드시 같을 필요는 없다. 암시적 변환 가능한 타입이면 된다.

2. Generic Container

Template 기반의 Single Linked List

```
template<typename T> struct Node
{
	T	data;
	Node*	next;
	Node(T a, Node* n) : data(a), next(n) {}
};
template<typename T> class slist
{
	Node<T>* head;
public:
	slist() : head(0) {}
	void push_front(const T& a) { head = new Node<T>(a, head); }
};
int main()
{
	slist<int> s;
	s.push_front(10);
	s.push_front(20);
	s.push_front(30);
	s.push_front(40);
	s.push_front(50);
}
```



main() 함수가 실행 되었을 때의 메모리 모양

slist 도 결국은 배열 같이 여러 개의 요소를 보관하는 컨테이너 이다. xfind()함수로 slist 에서 선형 검색을 할 수 있을까 ?

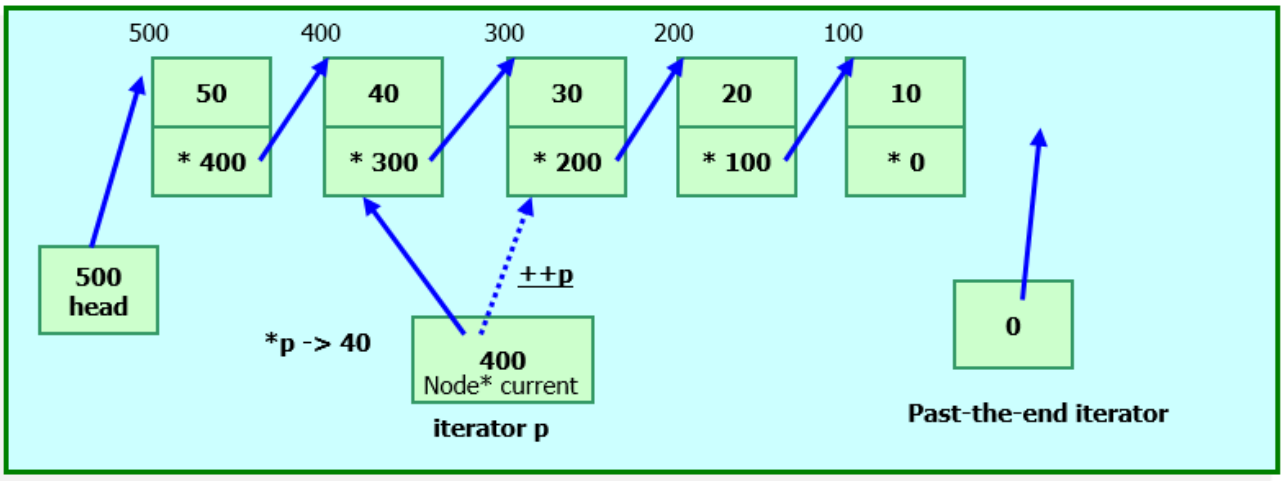
	xfind 함수	slist<>
다음 요소로 이동	++p	node = node->next
현재 위치에 있는 값 얻기	*p	node->data

xfind()는 결국 배열이라는 자료 구조에만 사용 가능하다. xfind()를 사용해서 slist 에서 검색을 할 수는 없을까 ?

3. 컨테이너와 알고리즘의 연결 - 반복자(iterator)

slist<> 안에 있는 Node를 가르 키는 스마트 포인터와 비슷한 역할의 객체.

++로 이동하고 *로 값을 꺼낼 수 있어야 하고 ==, != 연산이 사용가능 해야 한다.



iterator p는 진짜 포인터와 유사한 객체이다. ++, *를 사용 할 수 있다.

객체 형태의 컨테이너 버전은 상수 형 과 비상수형 함수를 모두 제공하고 있다.

```
template<typename T> class slist_iterator
{
	Node<T>* current;
public:
	slist_iterator(Node<T>* p = 0) : current(p) {}

	// xfind()에 전달 하려면 *, ++, ==, != 를 지원해야 한다.
	inline slist_iterator& operator++()
	{
		current = current->next;
		return *this;
	}
	inline T& operator*() { return current->data; }
	inline bool operator==(const slist_iterator& p)
	{ return current == p.current; }

	inline bool operator!=(const slist_iterator& p)
	{ return current != p.current; }
};
```

또한 모든 컨테이너 (slist)는 아래의 2 개 규칙을 지켜야 한다.

자신의 반복자 이름을 “iterator” 라는 약속된 형태로 외부에 노출해야 한다.

자신이 가진 요소의 처음과 마지막 다음을 가르키는 반복자를 리턴하는 begin(), end()함수를 제공해야 한다.

```
template<typename T> class slist
{
	Node<T>* head;
public:
	// 모든 컨테이너는 자신의 반복자 이름을 iterator 라는 약속된 이름으로 외부에 알려준다.
	typedef slist_iterator<T> iterator;

	slist() : head(0) {}
	void push_front(const T& a) { head = new Node<T>(a, head); }

	// 모든 컨테이너는 자신의 시작과 마지막 다음을 가르키는 반복자를 리턴하는 함수를 제공한다.
	iterator begin() { return iterator(head); }
	iterator end()   { return iterator(0); }
};
```

이제 xfind() 함수를 사용해서 slist<>에서 검색이 가능하다.

```
int main()
{
	slist<int> s;
	s.push_front(10);
	s.push_front(20);
	s.push_front(30);

	slist<int>::iterator p1 = xfind(s.begin(), s.end(), 20);
	cout << *p1 << endl;

	// 결국 p2는 추상화된 형태의 포인터이다.
	slist<int>::iterator p2 = s.begin();

	while (p2 != s.end())
	{
		cout << *p2 << endl;
		++p2;
	}
}
```

xfind()는 배열 뿐 아니라 모든 선형 컨테이너에서 값을 검색 할 수 있다.

4. C++ 표준 라이브러리 STL

Data를 보관하는 자료구조와 자료구조에 수행되는 연산을 분리한 라이브러리.

반복자(iterator)가 템플릿 기반 컨테이너와 일반화 알고리즘을 연결하는 역할을 한다.

```
#include <iostream>
#include <list>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
	string s = "hello";
	string::iterator p = s.begin();
	++p;
	cout << *p << endl; // 'e'

	reverse(s.begin(), s.end());  cout << s << endl; // "olleh"
	sort(s.begin(), s.end());  cout << s << endl; // ""

	list<int> st;
	st.push_front(10);
	st.push_front(20);
	st.push_front(30);


	reverse(st.begin(), st.end());

	list<int>::iterator p2 = st.begin();

	while (p2 != st.end())
	{
		cout << *p2 << endl;
		++p2;
	}
}
```

5. iterator category

수행 가능한 연산에 따른 반복자의 분류.

알고리즘에 전달되는 반복자의 종류를 구별하기 위해 사용

Iterator category	operation
입력 반복자(input iterator)	=*p, ++
출력 반복자(output iterator)	*p=, ++
전진형 반복자(forward iterator)	입출력, ++
양방향 반복자(bidirectional iterator)	입출력, ++, --
임의접근 반복자(random access iterator)	입출력, ++, --, +, -, []

컨테이너와 반복자 카테고리

Container	Iterater categories
일반 배열, vector, deque, string	임의 접근 반복자
list	양방향 반복자

일반화 알고리즘과 알고리즘이 요구하는 반복자 카테고리

Generic Algorithm	Iterater categories
find()	입력 반복자
reverse()	양방향 반복자
sort()	임의접근 반복자

slist<> 는 find()알고리즘은 사용할 수 있지만, reverse()알고리즘은 사용할 수 없다.

list<>는 reverse()알고리즘은 사용할 수 있지만, sort()알고리즘은 사용할 수 없다. 단, 멤버 함수 sort()가 있다.

vector<>에는 sort() 멤버 함수가 없다. vector는 모든 알고리즘을 사용할 수 있다.

```
int main()
{
	// single linked list
	slist<int> ss;
	find(ss.begin(), ss.end(), 20);		// ok
	reverse(ss.begin(), ss.end());		// error

	// double linked list
	list<int>  ds;
	reverse(ds.begin(), ds.end());		// ok
	sort(ds.begin(), ds.end());		// error
	ds.sort();				// ok

	// vector
	vector<int> v;
	sort(v.begin(), v.end());	// ok
	v.sort();			// error
}
```

임의의 반복자를 주어진 만큼 이동하는 advance() 알고리즘을 만들기

+ 연산을 사용한 반복자 이동

```
template<typename T> void xadvance( T& p, int n)
{
	p = p + n;
}
```

문제점 : 모든 반복자가 + 연산을 할 수 있는 것은 아니다.

++ 연산을 사용한 반복자 이동

모든 반복자는 ++ 연산은 가능하다.

```
template<typename T> void xadvance( T& p, int n)
{
	while( n--) ++p;
}
```

단점 : 임의접근 반복자는 ++ 보다 + 로 이동하는 것이 빠르다.

해결책

Iterator_category 를 타입화 한다.

모든 반복자 설계자는 자신의 반복자의 종류를 iterator_category 라는 약속된 이름으로 외부에 알려준다.

Iterator category 에 따라 함수 오버로딩으로 xadvance()를 구현한다.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag     :input_iterator_tag {};
struct bidirectional_iterator_tag:forward_iterator_tag {};
struct random_access_iterator_tag:bidirectional_iterator_tag{};

// 반복자 설계시 자신의 category를 알려준다.
template<typename T> class list_iterator
{
public: typedef bidirectional_iterator_tag iterator_categoty;
};
template<typename T> class vector_iterator
{
public: typedef random_access_iterator_tag iterator_categoty;
};
template<typename T> inline void xadvanceImp(T& p,int n,random_access_iterator_tag)
{
       cout << "임의접근" << endl; p = p + n;
}
template<typename T> inline void xadvanceImp( T& p, int n, input_iterator_tag)
{
       cout << "임의접근이 아닐때" << endl;  while( n--) ++p;
}
// 사용자가 사용하는 버전
template<typename T> inline void xadvance( T& p, int n)
{
       xadvanceImp( p, n, typename T::iterator_category() );
}
int main()
{
       int x[10] = { 1,2,3,4,5,6,7,8,9,10};
       vector<int> v(x, x+10);
       vector<int>::iterator p = v.begin();
       xadvance(p, 5);
       cout << *p << endl; // 6
}
```

6. 반복자 특질 (iterator traits)

문제점

일반 배열을 가르키는 포인터도 반복자 이다.

포인터 타입 안에는 iterator_category 라는 typedef 를 만들 수 없다.

```
template<typename T> inline void xadvance( T& p, int n)
{
	// T = int* 이므로 int*::iterator_category 가 된다. error
	xadvanceImp( p, n, typename T::iterator_category() );
}
int main()
{
	int x[10] = { 1,2,3,4,5,6,7,8,9,10};
	int* p = x; // p도 반복자 이다.
	xadvance(p, 5);
}
```

반복자 특질(iterator_traits)

반복자의 형태 : 진짜 포인터, 객체 형태로 만든 반복자

반복자의 형태에 상관없이 일반화된 알고리즘을 개발하기 위한 도구

```
template<typename T> struct iterator_traits
{
	typedef T::iterator_category iterator_category;
};
template<typename T> struct iterator_traits<T*>
{
	typedef random_access_iterator_tag iterator_category;
};
template<typename T> inline void xadvance( T& p, int n)
{
	xadvanceImp( p, n, typename iterator_traits<T>::iterator_category() );
}
int main()
{
	int x[10] = { 1,2,3,4,5,6,7,8,9,10};
	int* p = x;
	xadvance( p, 5);  // ok
	cout << *p << endl;  // 6
}
```

항목 5 - 07

멤버 함수가 아닌 begin(), end()

핵심개념

- 멤버 함수가 아닌 begin(), end() 함수가 필요한 이유
- 객체 형태의 컨테이너와 일반 배열에 모두 적용 가능한 알고리즘 만드는 방법
- C++ 표준이 제공하는 begin(), end() 함수의 원리
- range-base for 와 사용자 정의 컨테이너

1. 컨테이너의 모든 요소를 출력하는 show 함수

STL 의 다양한 컨테이너의 모든 요소를 출력하는 show 함수

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

// 컨테이너의 모든 요소를 출력하는 함수
template<typename T> void show(T& c)
{
	for ( auto p = c.begin(); p != c.end(); ++p)
	{
		cout << *p << endl;
	}
}
int main()
{
	vector<int> v = { 1,2,3,4,5,6,7,9,10 };
	show(v);

	int x[] = { 1,2,3,4,5,6,7,8,9,10 };
	show(x);  // error. x는 배열이므로 begin() 함수가 없다.
}
```

문제점 : x 는 배열 이므로 begin() 함수가 없다

2. 객체 형태의 컨테이너와 일반 배열에 모두 적용 가능한 show 만들기

해결책 1. is_array<> type traits 사용

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

// 컨테이너 버전
template<typename T> void showImpl(T& c, false_type)
{
	for (auto p = c.begin(); p != c.end(); ++p)
	{
		cout << *p << endl;
	}
```

```
}
// 배열 버전
template<typename T> void showImpl(T& c, true_type)
{
	for (auto p = c; p != c + extent<T, 0>::value ; ++p)
	{
		cout << *p << endl;
	}
}

// 컨테이너의 모든 요소를 출력하는 함수
template<typename T> void show(T& c)
{
	showImpl(c, is_array<T>());
}

int main()
{
	vector<int> v = { 1,2,3,4,5,6,7,9,10 };
	show(v);

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

문제점 : 컨테이너 버전과 배열 버전을 항상 별도의 함수로 만들어야 한다.

해결책 2. 멤버가 아닌 begin(), end() 함수

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

// 컨테이너 버전
template<class _Container>
inline auto  begin(_Container& _Cont) -> decltype(_Cont.begin())
{
	return (_Cont.begin());
}
template<class _Container>
inline auto end(_Container& _Cont) -> decltype(_Cont.end())
{
	return (_Cont.end());
}
// 배열 버전
```

```
template<class _Ty, size_t _Size> inline _Ty* begin(_Ty(&_Array)[_Size]) noexcept
{
	return (_Array);
}
template<class _Ty, size_t _Size> inline _Ty* end(_Ty(&_Array)[_Size]) noexcept
{
	return (_Array + _Size);
}
// 컨테이너의 모든 요소를 출력하는 함수
template<typename T> void show(T& c)
{
	for (auto p = begin(c); p != end(c); ++p)
		cout << *p << endl;
}
int main()
{
	vector<int> v = { 1,2,3,4,5,6,7,9,10 };
	show(v);
	int x[] = { 1,2,3,4,5,6,7,8,9,10 };
	show(x);  // error. x는 배열이므로 begin() 함수가 없다.
}
```

장점 : 하나의 일반화 함수(show)로 객체 형태의 컨테이너와 일반 배열에 모두 사용 가능한 함수를 만들 수 있다.

3. C++11/14 표준이 제공하는 begin(), end()함수의 모양

Defined in header <iterator>

객체 형태의 컨테이너 버전은 상수 형 과 비상수형 함수를 모두 제공하고 있다.

```
// 컨테이너 버전
template<class _Container>
inline auto  begin(_Container& _Cont) -> decltype(_Cont.begin())
{
	return (_Cont.begin());
}

template<class _Container>
inline auto end(_Container& _Cont) -> decltype(_Cont.end())
{
	return (_Cont.end());
}
```

```
// 컨테이너 상수 버전
template<class _Container>
inline auto begin(const _Container& _Cont) -> decltype(_Cont.begin())
{
	return (_Cont.begin());
}

template<class _Container>
inline auto end(const _Container& _Cont) -> decltype(_Cont.end())
{
	return (_Cont.end());
}

// 배열 버전
template<class _Ty,  size_t _Size> inline _Ty* begin(_Ty(&_Array)[_Size]) noexcept
{
	return (_Array);
}

template<class _Ty,  size_t _Size> inline _Ty* end(_Ty(&_Array)[_Size]) noexcept
{
	return (_Array + _Size);
}
```

4. 사용자 정의 타입과 range-base for

사용자 정의 타입에 range-base for문을 사용할 수 있을까 ?

```
template<typename T> struct Point3D
{
	T x, y, z;
};
int main()
{
	Point3D<int> p1 = { 1,2,3 };

	for (auto& n : p1)  // ?
		cout << n << endl;
}
```

일반 함수 **begin(), end()를 사용자 정의 타입에 대해서 오버로딩** 하면 range-base for 를 사용할 수 있다.

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

```
template<typename T> struct Point3D
{
	T x, y, z;
};

// ranged-based for 를 사용하려면 begin()/end() 함수를 제공해야 한다.
template<typename T> inline T* begin(Point3D<T>& p) noexcept
{
	return &(p.x);
}
template<typename T> inline T* end(Point3D<T>& p) noexcept
{
	return &(p.z) + 1;
}
template<typename T> inline const T* begin(const Point3D<T>& p) noexcept
{
	return &(p.x);
}
template<typename T> inline const T* end(const Point3D<T>& p) noexcept
{
	return &(p.z) + 1;
}
int main()
{
	Point3D<int> p1 = { 1,2,3 };

	for (auto& n : p1)
	{
		cout << n << endl;
	}

	for (const auto& n : p1)
	{
		cout << n << endl;
	}
}
```

주의 ! 상수객체 버전과 비 상수 객체 버전을 모두 제공해야 한다.