제4장 포함과 상속

1. 포함 (composition)

- 클래스를 재활용하는 개념
- 클래스내의 멤버로 다른 클래스의 객체를 사용
- 포함된 개체를 멤버데이터로 소유하고 있는 클래스를 둘러싼 클래스(Surrounded class)라고 한다.
- 둘러싼 클래스로부터 개체를 만들 경우, 둘러싼 클래스의 생성자함수 보다 포함된 객체의 생성자 함수가 먼저 수행됨.

  ** 복합객체

[예제4-1] 포함의 예

(Profile.h)

```
#pragma once
#include <iostream>
using namespace std;
#include "MyString.h"

class Profile
{
private:
        MyString name;
        MyString addr;
        int age;
        double height;
public :
        Profile();
        Profile(const char *np, const char *ap, int a, double h);
        bool operator>(const Profile &br);
        MyString getName();
        MyString getAddr();
        int getAge();
        double getHeight();
        void setName(MyString &br);
        void setAddr(MyString &br);
        void setAge(int a);
        void setHeight(double h);
};
```

(Profile.cpp)

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

```
Profile::Profile() : name(), addr(), age(0), height(0)
{ }
//-------------------------------------------------------------------------------
Profile::Profile(const char *np, const char *ap, int a, double h)
: name(np), addr(ap), age(a), height(h)
{ }
//-------------------------------------------------------------------------------
bool Profile::operator>(const Profile &br)
{
	if(height>br.height) return true;
	else return false;
}
//-------------------------------------------------------------------------------
MyString Profile::getName()
{
	return name;
}
//-------------------------------------------------------------------------------
MyString Profile::getAddr()
{
	return addr;
}
//-------------------------------------------------------------------------------
int Profile::getAge()
{
	return age;
}
//-------------------------------------------------------------------------------
double Profile::getHeight()
{
	return height;
}
//-------------------------------------------------------------------------------
void Profile::setName(MyString &br)
{
	name=br;
}
//-------------------------------------------------------------------------------
void Profile::setAddr(MyString &br)
{
	addr=br;
}
//-------------------------------------------------------------------------------
void Profile::setAge(int a)
{
	age=a;
}
//-------------------------------------------------------------------------------
void Profile::setHeight(double h)
{
	height=h;
}
```

(MyString.h)

```
#pragma once
```

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

class MyString
{
private:
	char *str;   // 문자열을연결할포인터멤버
	int len;   // 문자열의길이를저장
public:
	MyString();   // 디폴트생성자
	MyString(const char *cp);   // 오버로디드생성자
	MyString(const MyString &br);   // 복사생성자
	~MyString();   // 소멸자
	MyString &operator=(const MyString &br);   // 대입연산자멤버함수
	MyString operator+(const MyString &br);   // 덧셈연산자
	bool operator>(const MyString &br);   // 관계연산자멤버함수
	char * getStr();
	int getLen();
	void setStr(char *cp);
	void setLen(int n);
};
```

(MyString.cpp)

```
#pragma warning (disable:4996)
#include <iostream>
#include <string.h>
#include "MyString.h"
using namespace std;

MyString::MyString()
{
	len=0;
	str=new char[len+1];
	strcpy(str, "");
}
//-----------------------------------------------------------------------------
MyString::MyString(const char *cp)
{
	len=strlen(cp);
	str=new char[len+1];
	strcpy(str, cp);
}
//-----------------------------------------------------------------------------
MyString::MyString(const MyString &br)
{
	len=br.len;
	str=new char[len+1];
	strcpy(str, br.str);
}
//-----------------------------------------------------------------------------
MyString::~MyString()
{
	delete [] str;
}
//-----------------------------------------------------------------------------
```

```
MyString &MyString::operator=(const MyString &br)
{
	if(this==&br) return *this;
	len=br.len;
	delete [] str;
	str=new char[len+1];
	strcpy(str, br.str);
	return *this;
}
//-----------------------------------------------------------------------------
MyString MyString::operator+(const MyString &br)
{
	int ln=len+br.len;
	char *tp=new char[ln+1];
	strcpy(tp, str);
	strcat(tp, br.str);
	MyString temp(tp);
	delete[] tp;
	return temp;
}
//-----------------------------------------------------------------------------
bool MyString::operator>(const MyString &br)
{
	if(len>br.len) return true;
	else return false;
}
//-----------------------------------------------------------------------------
char * MyString::getStr()
{
	return str;
}
//-----------------------------------------------------------------------------
int MyString::getLen()
{
	return len;
}
//-----------------------------------------------------------------------------
void MyString::setStr(char *cp)
{
	if(this->str==cp) return; // 내 문자열을 내가 저장하려고 하면 리턴
	len = strlen(cp);
	delete []str;
	str=new char[len+1];
	strcpy(str, cp);
}
//-----------------------------------------------------------------------------
void MyString::setLen(int n)
{
	len = n;
}
```

(main.cpp)

```
#include <iostream>
#include <string.h>
#include "Profile.h"
```

```
#include "MyString.h"
using namespace std;

void userScreen(void);
char menu(void);
int proInput(Profile *);
void proDisplay(Profile *,int);

int main()
{
	userScreen();
	return 0;
}
//-----------------------------------------------------------------------------------------
void userScreen()
{
	Profile  ary[20];    // 프로필 저장 객체 배열
	int profileCnt;      // 입력 받은 프로필의 개수 카운트 변수
	char ch;             // 메뉴번호 저장 변수

	while(1)
	{
		ch=menu();          // 입력된 메뉴의 번호를 ch에 리턴받음
		if(ch=='4') {break;}  // 4번 종료하기 메뉴 입력시 무한반복문을 탈출
		switch(ch)
		{
			case '1' : profileCnt = proInput(ary); break;
			//case '2' : proUpdate(ary,profileCnt); break;
			case '3' : proDisplay(ary,profileCnt); break;
		}
	}
}
//-----------------------------------------------------------------------------------------
char menu(void)
{
	char n=0, i;
	char menulist[4][20] = { "프로필 입력", "프로필 수정", "프로필 출력","종료" };

	for(i=0; i<4; i++)    // 메뉴 문자열을 출력하는 반복문
	{
		cout << i+1 <<". " << menulist[i]<< endl;
	}

	while(n<'1' || n>'4') // 유효한 번호가 아니면 다시 입력
	{
		cout << "* Select menu No. : _₩b";
		cin >> n;
	}
	return(n);
}
 //-----------------------------------------------------------------------------------------
int proInput(Profile *ap)
{
	char name[50];
	char addr[80];
```

```
        int age;
        double height;
        int cnt=0;  // 입력 받은 프로필의 개수를 카운트하는 변수

        while(1)
        {
                cout << "이름입력 : ";
                cin >> name;
                if(strcmp(name,"끝")==0) {break;}
                cout << "주소입력 : ";              cin >> addr;
                cout << "나이입력 : ";              cin >> age;
                cout << "키입력 : ";                cin >> height;

                ap[cnt]=Profile(name, addr, age, height); // 입력 받은 데이터로 객체 초기화
                cnt++;
        }
        return cnt;
}
//------------------------------------------------------------
void proDisplay(Profile *ap, int cnt)
{
        int i;
        for(i=0; i<cnt; i++)
        {
             cout << i+1 <<". " << ap[i].getName().getStr() << " : "
             << ap[i].getAddr().getStr() << "/" << ap[i].getAge() << "세/"
             << ap[i].getHeight() << "cm" << endl;
        }
}
```

* 2번 메뉴 "프로필 수정" 기능(proUpdate() 함수)을 완성해 보시오.

2. 상 속 (inheritance)

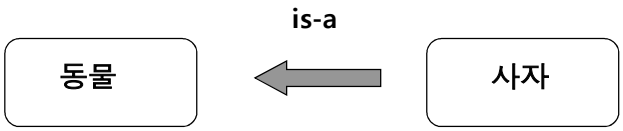
- 기존 클래스의 기능이나 특성을 재활용하는 방법에는 포함 외에 상속이 있다.
- 기존의 클래스 코드를 재활용한다는 입장에서 객체지향 프로그래밍에서 매우 유용
- 물려주는 클래스를 기본(base)클래스라 하고, 물려받는 클래스를 파생(derived)클래스라 한다.
- 파생클래스에서는 기본 클래스의 코드를 그대로 활용하고 새롭게 추가할 부분만 정의하거나 수정하면 되기 때문에 코드의 재활용 측면과 개발 기간 단축에 큰 효과가 있다.

```
class 파생 class 명 : 파생유형 기본 class 명
{
        멤버 1;
        멤버 2;
        ...
};
```

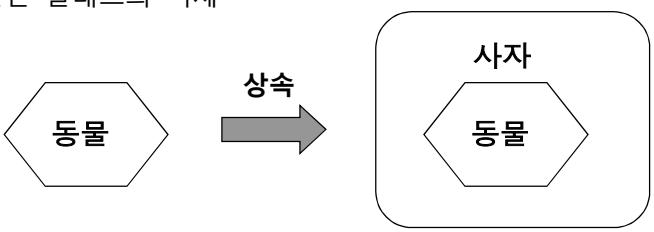
· 계승시키는 클래스 : 기본(base), 상위(super), 부모(parent) 클래스
· 계승받는 클래스 : 파생(derived), 하위(sub), 자식(child) 클래스
· 직접접근 가능하게 상속되는 것 : 멤버데이타, 멤버함수 (protected, public)
· 직접접근 불가능하게 상속되는 것 : 생성자 함수, 소멸자 함수,
private 멤버데이타, private 멤버함수
· 객체생성 : 상위클래스에서 하위클래스 순서로 만들어진다.
. 객체소멸 : 하위클래스에서 상위클래스 순서로 소멸된다.

◈ is-a 관계

- 상속관계에 있는 두 클래스 사이의 관계는 is-a관계라 함



- 상속 받은 클래스의 객체



◈ 상속지정자(파생유형)

클래스가 상속될 때 파생클래스에서 기본클래스의 멤버들에 대한 접근 권한이 지정되는데 이를 파생유형이라 하며 public, protected, private의 파생유형을 갖는다.
대부분의 public 파생유형으로 상속받아 처리한다.

((파생유형에 따른 멤버 접근 권한))

파생유형 / 접근지정자	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	직접 접근 불가		

[예제4-2] 상속의 예

(MoneyBox.h)

```
// 기본클래스(MoneyBox) 선언
#pragma once
#include "MyString.h"
class MoneyBox{
private:
        MyString name;   // 예금주
        int sum;     // 잔액
public:
        MoneyBox(const char *np="아무개", int m=0); // 오버로디드생성자
        MoneyBox(const MoneyBox &br);   // 복사생성자
        void save(int m);          // 저금
        void use(int m);           // 사용
        void setName(MyString name);     // name멤버수정
        MyString getName();                // name멤버값리턴
        void setSum(int sum);             // sum멤버수정
        int getSum();                      // sum멤버값리턴
};
```

(MoneyBox.cpp)

```
// MoneyBox 멤버함수의정의(MoneyBox.cpp)
#include <iostream>
using namespace std;
#include <string.h>
#include "MoneyBox.h"
//---------------------------------------------------------------------
MoneyBox::MoneyBox(const char *name, int sum) : name(name) //
멤버명과전달인자명이같은데괜찮은가?
{
        this->sum=sum;
}
```

```
//---------------------------------------------------------------------
MoneyBox::MoneyBox(const MoneyBox &br) : name(br.name)
{
	this->sum=br.sum;
}
//---------------------------------------------------------------------
void MoneyBox::save(int money)
{
	if(money>=0)
	{
		this->sum += money;
	}
}
//---------------------------------------------------------------------
void MoneyBox::use(int money)
{
	if(money >= 0 && money <= this->sum)
	{
		this->sum -= money;
	}
}
//---------------------------------------------------------------------
void MoneyBox::setName(MyString name)
{
	this->name = name;
}
//---------------------------------------------------------------------
MyString MoneyBox::getName()
{
	return this->name;
}
//---------------------------------------------------------------------
void MoneyBox::setSum(int sum)
{
	this->sum = sum;
}
//---------------------------------------------------------------------
int MoneyBox::getSum()
{
	return this->sum;
}
//---------------------------------------------------------------------
```

(iMoneyBox.h)

```
// 파생클래스(iMoneyBox) 선언
#pragma once
#include "MoneyBox.h"

class iMoneyBox : public  MoneyBox
{
private:
	int limit;  // 저금통의한계금액을저장하기위해추가된데이터멤버
public:
	iMoneyBox(const char *np="아무개", int m=0, int l=1000); // 생성자
	void setLimit(int limit);    // limit 멤버수정
```

```
        int getLimit();                 // limit 멤버의값리턴
        int check();                    // 현재사용량검사
};
```

(iMoneyBox.cpp)

```
// iMoneyBox의선언, 멤버함수의정의 (iMoneyBox.cpp)
#include <iostream>
using namespace std;
#include "imoneybox.h"

iMoneyBox::iMoneyBox(const char *np, int m, int l) : MoneyBox(np, m)
{
        this->limit=l;
}
//---------------------------------------------------------------------
int iMoneyBox::check()
{
        int temp;
        temp=int((double)this->getSum()/this->limit*100);
        return temp;
}
//---------------------------------------------------------------------
void iMoneyBox::setLimit(int limit)
{
        this->limit = limit;
}
//---------------------------------------------------------------------
int iMoneyBox::getLimit()
{
        return this->limit;
}
```

(main.cpp)

```
#include<iostream>
using namespace std;
#include "iMoneyBox.h"
int main()
 {
        iMoneyBox a("철이", 100, 2000), b("메텔", 500, 3000);

        a.save(600);
        b.use(100);
        cout<<a.getName().getStr()<<" " <<a.getSum() << " " << a.getLimit() << endl;
        cout<<b.getName().getStr()<<" " <<b.getSum() << " " << b.getLimit() << endl;
        cout << "철이의저금통사용량: " << a.check() << '%' << endl;
        cout << "메텔의저금통사용량: " << b.check() << '%' << endl;
        return 0;
 }
```

3. 멤버함수의 재정의 (overriding)

- 기본클래스의 멤버함수를 파생클래스에 그대로 사용하지 않고 재정의하여 기능확장 또는 기능변경하여 사용하는 기능

- 기본클래스의 멤버함수를 overriding 하더라도 파생클래스 안에서는 여전히 사용 가능하다.
  예) 기본클래스명::멤버함수명();

- overriding되는 멤버함수는 함수명은 물론 나머지 시그니처까지 완전히 일치해야 한다.
  즉, 기본클래스의 멤버함수가 3개로 overloading되어 있다면 파생클래스에서도 그 중 하나만 또는 3개 모두를 overriding할 수 있다.

예) 위에서 살펴본 iMoneyBox 파생클래스에 save와 같은 이름의 멤버함수를 추가한다.

```
class iMoneyBox : public MoneyBox
{
private:
    int limit;
public:
    iMoneyBox(const char *np="아무개", int m=0, int l=1000); // 생성자
    void save(int m);  // 멤버함수의 재정의 선언
    int check();  // 현재 사용량 검사
    friend ostream &operator<<(ostream &os, const iMoneyBox &br);
};
// save멤버함수의 재정의
void iMoneyBox::save(int m)
{
    if(m>limit-getSum()){
        cout << "저금통의 공간이 부족합니다..." << endl;
        return;
    }
    MoneyBox::save(m);  // 상속 받은 기본클래스의 save()함수 사용
}
```

[예제4-3] 재정의의 예

```
1:  #include <iostream>
2:  using namespace std;
3:  // 기본클래스
4:  class Base
5:  {
6:  public:
7:      void func(char ch) { cout << "Base(char) : " << ch << endl; }
8:      void func(int in) { cout << "Base(int) : " << in << endl; }
9:  };

10: // 재정의가 없는 파생클래스
11: class Derived : public Base {};

12: // 재정의가 있는 파생클래스
13: class rDerived : public Base
14: {
15: public:
16:     void func(char ch)
17:     { cout << "rDerived(char) : " << ch << endl; }
18: };
19:
20: int main()
21: {
22:     Derived a;
23:     cout << "재정의 없는 파생클래스..." << endl;
24:     a.func('A');
25:     a.func(10);
26:
27:     rDerived b;
28:     cout << "재정의한 파생클래스..." << endl;
29:     b.func('B');
30:     b.func(97);
31:     b.func(100.5);
32:
33:     return 0;
34: }
```

◈ 기본클래스와 파생클래스의 포인터

• 파생클래스는 비록 기본클래스로부터 상속 받지만 기본적으로 기본클래스와는 다른 자료형으로 인식된다. 그러나 C++은 기본클래스의 포인터나 참조객체를 통해 파생클래스의 객체를 참조할 수 있도록 함으로서 두 객체를 모두 처리할 수 있다.

[예제4-4] 객체를 가리키는 포인터변수로 멤버함수 호출

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

class Base{   // 기본클래스
        int va;
  public:
        Base() { va=10;}
        void view() { cout << "Base view..." <<va<< endl; }
};
class Derived : public Base{   // 파생클래스
        int vb;
  public:
        Derived() { vb = 22; }
        void view()   // view함수의 재정의
        { cout << "Derived view..." << vb << endl; }
};
int main()
{
        Base a;
        Derived b;

        Base *bp=&a;
        Derived *dp=&b;
        bp->view();   // 기본클래스의 view호출
        dp->view();   // 파생클래스의 재정의된 view호출

        bp = &b;       // 상향 캐스팅(up-casting) - 자동(암시적) 형변환 가능
        bp->view();   // 기본클래스의 view()호출

        dp = (Derived *)&a;// 하향 캐스팅(down-casting) - 강제(명시적) 형변환 필수
        dp->view(); // 파생클래스의 view()호출 (vb멤버의 자리에는 무엇이 출력될까?)

        return 0;
}
```

• 상항 캐스팅(up-casting)

기본클래스의 포인터 변수로 파생클래스의 객체를 가리키게 하는 것.

별도의 캐스팅 없이도 가능

이때 포인터는 파생클래스 내의 기본클래스 객체 영역을 가리키게 된다.

예)

```
Derived b;
Base *bp=&b;  // 기본클래스의 포인터변수에 파생클래스 객체의 연결
bp->view();  // 기본클래스 view가 호출된다. "Base view..." 출력
```

• 하향 캐스팅(down-casting)

파생클래스의 포인터 변수로 기본클래스의 객체를 가리키게 하는 것.

강제형변환 할 경우에만 가능하다.

예)

```
Base *bp;
Derived a, *dp;
bp=&a;
bp->view();  // "Base view..." 출력
dp=(Derived *)bp;  // 명시적인 형변환(하향 캐스팅)
dp->view();  // "Derived view..." 출력
```

4. 가상(virtual)

◈ 바인딩(binding)의 이해

- 바인딩(binding) : 함수의 이름과 함수의 구현부를 결합
  특정 함수 호출 시, 특정 함수의 구현에 연결하는 것

- 정적 바인딩(static binding)
  컴파일러가 컴파일할 때 객체의 포인터나 참조객체를 통해 접근되는 클래스의 멤버함수를 자신의 타입으로 인식하고 일치시키는 형태로 수행되는 바인딩

- 동적 바인딩(dynamic binding or late binding)
  컴파일러가 사용할 클래스의 멤버 함수를 컴파일 타임에 결정하지 않고
  프로그램 실행 중에 호출되어 수행될 클래스의 멤버함수를 결정하게 하는 바인딩

※ 클래스의 멤버 함수에 대해서만 사용 가능

[예제4-5] 정적 바인딩의 예

```
#include<iostream>
using namespace std;
class Animal
{
public:
	void show()
	{
		cout << "동물입니다.₩n";
	}
};
class Cat : public Animal
{
public:
	void show()
	{
		cout << "고양이입니다.₩n";
	}
};
int main()
{
	Animal a_ob;
	Cat c_ob;
	a_ob.show();
	c_ob.show();

	Animal * a_ptr;
	a_ptr = &a_ob;
	a_ptr->show();
	a_ptr = &c_ob;
	a_ptr->show();
	return 0;
}
```

◈ 가상(virtual)의 개념

- 멤버 함수 호출에 필요한 함수의 호출 번지나 클래스의 참조가 컴파일 타임에 결정되는 것이 아니라 실행중에 이루어지는 것을 의미 (동적 바인딩)

- 다형성을 지원해 주는 키워드 : 동일한 인터페이스로 다양한 동작을 수행할 수 있음

◈ 가상함수(virtual function)

- 기본클래스의 포인터 변수로 파생클래스의 overriding된 함수를 사용하려면 기본클래스의 멤버함수는 가상함수로 지정되어있어야 한다.

- 기본 클래스의 멤버함수에 virtual 키워드를 사용하여 파생클래스의 멤버 함수로 연결하는 개념.

[예제4-6] 가상함수를 사용한 프로그램

```
1:    #include <iostream>
2:    using namespace std;
3:    class Base
      {
4:    public:
5:          virtual void view() { cout << "Base view..." << endl; }
6:    };
7:
8:    class Derived : public Base
9:    {
10:   public:
11:         virtual void view() { cout << "Derived view..." << endl; } // 파생클래스의 virtual
12:                                                                    // 키워드는 생략 가능함
13:   };
14:
15:   int main()
16:   {
17:         Base a, *bp;
18:         Derived b;
19:
20:         bp=&a;       // 기본클래스의 객체 참조
21:         bp->view();  // 기본클래스의 view함수 호출
22:         bp=&b;       // 파생클래스의 객체 참조
23:         bp->view();  // 파생클래스의 view함수 호출
24:
25:         return 0;
26:   }
```

[예제4-7] 가상함수를 사용한 다형성의 구현

```
1:  #include <iostream>
2:  using namespace std;
3:  #include <string.h>
4:  class Teacher{
5:  private:
6:      char name[20];
7:      int pay;
8:  public:
9:      Teacher(const char *np="아무개", int p=0){
10:         strcpy(name, np);
11:         pay=p;
12:     }
13:     virtual void teach() { cout << "가르친다..." << endl; }
14:     void view(){
15:         cout << "이름 : " << name << ", 수업료 : " << pay << endl;
16:     }
17: };
18:
19: class kTeacher : public Teacher{
20: public:
21:     kTeacher(const char *np="아무개", int p=0) : Teacher(np, p) {}
22:     void teach() { cout << "국어를 가르친다..." << endl; }
23: };
24:
25: class eTeacher : public Teacher{
26: public:
27:     eTeacher(const char *np="아무개", int p=0) : Teacher(np, p) {}
28:     void teach() { cout << "영어를 가르친다..." << endl; }
29: };
30:
31: class mTeacher : public Teacher{
32: public:
33:     mTeacher(const char *np="아무개", int p=0) : Teacher(np, p) {}
34:     void teach() { cout << "수학을 가르친다..." << endl; }
35: };
36:
37: int main()
38: {
39:     Teacher *cheoli[2];
40:     cheoli[0]=new kTeacher("홍길동", 70);
41:     cheoli[1]=new mTeacher("이순신", 80);
42:
43:     Teacher *metel[3];
44:     metel[0]=new kTeacher("신사임당", 200);
45:     metel[1]=new eTeacher("임꺽정", 50);
46:     metel[2]=new mTeacher("강감찬", 90);
47:
48:     int i;
49:     cout << "철이의 선생님..." << endl;
50:     for(i=0; i<2; i++){
51:         cheoli[i]->view();
52:         cheoli[i]->teach();
53:     }
```

```
54:
55:       cout << "메텔의 선생님..." << endl;
56:       for(i=0; i<3; i++){
57:            metel[i]->view();
58:            metel[i]->teach();
59:       }
60:
61:       return 0;
62: }
```

◈ 가상 소멸자(virtual desturctor)

· 상속 관계에서 기본클래스의 포인터 변수로 파생클래스의 객체에 접근하는 경우, 기본클래스의 소멸자가 호출되어 기본클래스의 객체로 할당된 메모리는 정상적으로 제거 되지만 파생 클래스의 메모리는 정상적으로 제거 되지 않는 문제점이 발생한다.
· 파생클래스의 소멸자가 정확히 호출되도록 기본 클래스의 소멸자를 가상 함수로 정의 해야 함

[예제4-8] 가상 소멸자

```
#include <iostream>
using namespace std;
class BaseClass
{
public:
    BaseClass() { cout << "[BaseClass] 생성자\n\n"; }
    virtual ~BaseClass() { cout << "[BaseClass] 소멸자\n\n"; }
};
class DerivedClass : public BaseClass
{
public:
    DerivedClass() { cout << "[DerivedClass] 생성자\n\n"; }
    ~DerivedClass() { cout << "[DerivedClass] 소멸자\n\n"; }
};
int main()
{
    BaseClass * base = new DerivedClass;
    delete base;
    return 0;
}
```

5. 추상클래스(abstract class)

- 순수가상함수(가상함수 선언부 뒤에 "=0"을 넣어 만 듬, 정의부를 갖지 않음)를 포함하고 있는 클래스를 추상클래스라 한다.
- 추상클래스는 객체를 생성할 수 없다.
- 추상클래스는 다형성을 위한 상속의 기본클래스로 사용하기 위해서 설계한다.

예)

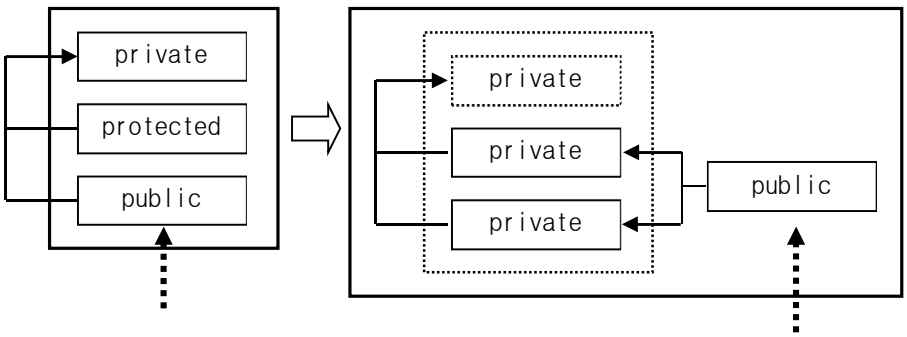
```
class Teacher
{
private:
    char name[20];
    int pay;
public:
    Teacher(const char *np="아무개", int p=0)
    {
        strcpy(name, np);
        pay=p;
    }
    virtual void teach() =0;  // 순수가상함수 선언 => 추상클래스가 된다.
    void view()
    {
        cout << "이름 : " << name << ", 수업료 : " << pay << endl;
    }
};
Teacher a;  // 독자적으로 객체를 생성할 수 없다.
```

6. public 이외의 상속과 다중상속

▌private과 protected 상속

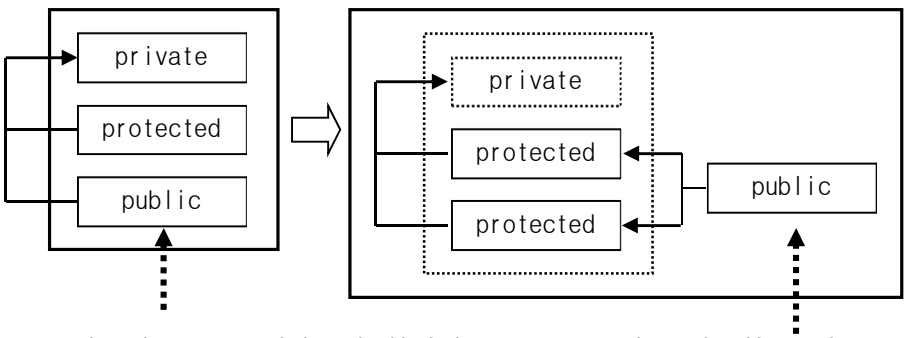
private 상속은 기본클래스의 protected와 public 접근권한을 가진 멤버가 모두 파생클래스의 private 멤버로 상속되는 것이다. 따라서 파생클래스 안에서는 자유롭게 접근할 수 있지만 외부로부터의 접근은 차단 된다. 기본클래스의 private멤버는 public상속과 마찬가지로 파생클래스 안에서조차 직접 접근이 불가능하다.

기본 클래스 (**private 상속**) 파생 클래스



결국 기본클래스의 구성요소는 가지고 가되 인터페이스 역할을 하는 public 멤버함수를 파생클래스에서는 사용하지 않겠다는 뜻이다. 이것은 다른 클래스의 객체를 멤버로 포함시키는 재활용기법과 유사한 구조를 가진다. 다만 파생클래스에서 상속된 부분을 식별할 이름이 없으므로 파생클래스의 구현이 좀더 불편하다는 것이다.
protected 상속도 마찬가지이다. 기본클래스의 protected와 public멤버를 모두 protected멤버로 상속한다. protected 접근권한 역시 외부로부터의 접근이 불가능하므로 기본클래스의 인터페이스를 파생클래스에서 사용할 수 없다.

기본 클래스 (**protected 상속**) 파생 클래스



private상속과 protected상속의 차이점은 protected로 상속하는 경우는 파생클래스가 재상속될 때 파생클래스의 protected멤버를 재상속한 파생클래스에서 직접 접근할 수 있다.

반면에 private상속은 기본클래스의 멤버가 private으로 상속되기 때문에 이후에 재상속된 파생클래스에서는 직접 접근이 불가능하게 된다. 결국 기본클래스 멤버의 접근권한을 바로 다음에 상속하는 클래스에게만 주겠다는 의도이다.

private과 protected상속은 public멤버함수의 재사용을 차단하므로 상속이라기보다 **포함**의 성격이

강하다. 따라서 멤버함수의 상속이 불필요한 경우라면 포함의 방법을 선택하는 것이 바람직하다. 다만 기본클래스의 protected멤버가 파생클래스에서 빈번하게 사용된다면 private상속을 통해 기본클래스의 멤버함수는 차단하면서 그 멤버는 파생클래스에서 자유롭게 사용하도록 만들 수 있다.

[예제4-9] private상속을 사용한 프로그램

```
1:  #include <iostream>
2:  using namespace std;
3:  #include <string.h>
4:  // 기본클래스의 선언
5:
6:  class Player{
7:  protected:
8:      char name[20];  // 이름
9:      int salary;  // 연봉
10:     double winning;  // 승률
11: public:
12:     Player(const char *np="아무개", int s=100, double w=0);
13:     void exercise();  // 연습
14:     void play();  // 경기
15:     void view() const;  // 데이터 출력
16: };
17:
18: // 기본클래스의 멤버함수 정의
19:
20: Player::Player(const char *np, int s, double w)
21: {
22:     strcpy(name, np);
23:     salary=s;
24:     winning=w;
25: }
26:
27: void Player::exercise() { cout << "선수가 연습한다..." << endl; }
28: void Player::play() { cout << "선수가 경기한다..." << endl; }
29:
30: void Player::view() const
31: {
32:     cout << "선수 이름 : " << name << endl;
33:     cout << "선수 연봉 : " << salary << endl;
34:     cout << "선수 승률 : " << winning << '%' << endl;
35: }
36:
37: // 파생클래스의 선언(private상속)
38:
39: class Manager : private Player{
40: public:
41:     Manager(const char *np="아무개", int s=100, double w=0);
42:     void command();
43:     void view() const;
44: };
45:
46: // 파생클래스 멤버함수의 정의
47:
48: Manager::Manager(const char *np, int s, double w)
```

```
49: {
50:         strcpy(name, np);   // 기본클래스의 멤버에 직접 접근할 수 있다.
51:         salary=s;
52:         winning=w;
53: }
54:
55: void Manager::command()
56: { cout << "감독이 지시를 내린다..." << endl; }
57:
58: void Manager::view() const
59: {
60:         cout << "감독 이름 : " << name << endl;
61:         cout << "감독 연봉 : " << salary << endl;
62:         cout << "감독 승률 : " << winning << '%' << endl;
63: }
64:
65: int main()
66: {
67:         Player a("홍길동", 2000, 70.0);
68:
69:         a.exercise();
70:         a.play();
71:         a.view();
72:
73:         Manager b("이순신", 5000, 90.0);
74:         b.command();
75:         b.view();
76:         //   b.play();   => 에러!
77:         //   b.exercise();   => 에러!
78:
79:         Player * p = (Player *)&b;   // up-casting 이 자동으로 이루어지지 않음
80:        p->view();
81:        return 0;
82 }
```

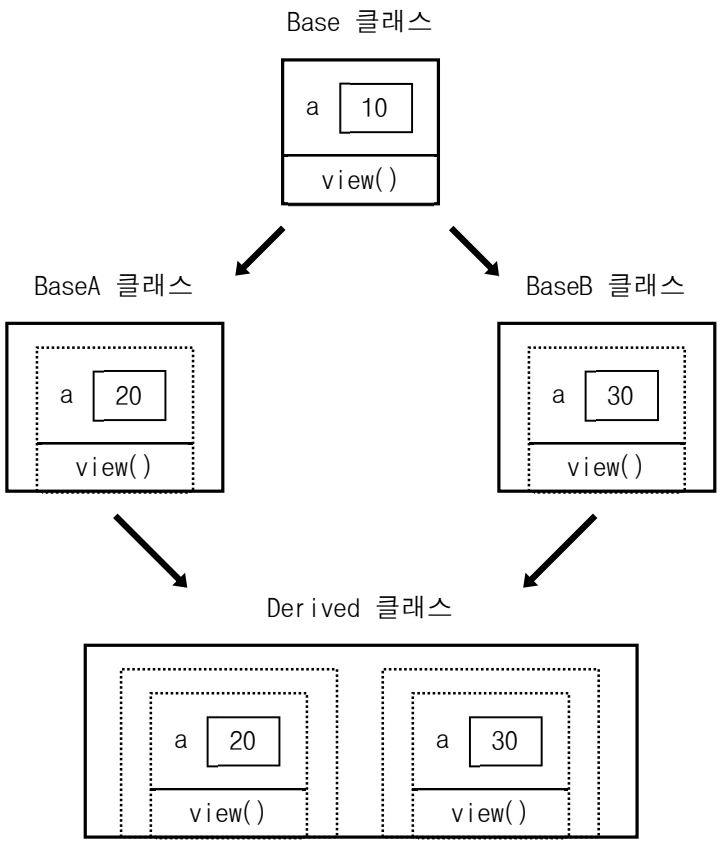
▌다중 상속(Multiple Inheritance)

두 개 이상의 클래스로부터 상속받아 새로운 클래스를 정의하는 것을 다중 상속이라 한다.
상속 받는 클래스는 개별적으로 상속모드를 사용할 수 있으며, 여러 기본클래스의 멤버를 동시에 상속받아 클래스를 쉽게 확장할 수 있다. 그러나 상속경로가 달라 여러 가지 복잡한 문제를 일으키므로 지극히 제한적으로 사용해야 한다. 먼저 다중상속의 문제점을 살펴보고 해결방법을 찾아보자.

다중상속의 가장 큰 문제는 상속 되는 두 개 이상의 기본클래스가 공통의 기본클래스를 갖는 경우 다중 상속하는 파생클래스에서 공통의 기본클래스가 중복되는 것이다.

```
class Base{
private:
    int a;
public:
    Base(int n=10) { a=n; }   // 생성자
```

```
    void view() { cout << "Base(" << a << ')' << endl; } // 출력 멤버함수
};

class BaseA : public Base {
public:
    BaseA() : Base(20) {}   // 기본클래스를 초기화하는 디폴트생성자
};

class BaseB : public Base {
public:
    BaseB() : Base(30) {}   // 기본클래스를 초기화하는 디폴트생성자
};

class Derived : public BaseA, public BaseB {};   // 다중상속

int main()
{
    Derived d;
    d.view();   // view()가 두 개이므로 호출이 모호해서 에러 발생

    return 0;
}
```



Derived클래스의 객체는 다중상속을 통해 중복된 Base클래스의 멤버들을 가지게 된다. 물론 데이터부분은 private멤버이고 여러 값을 가져야 할 필요도 있으므로 중복이 항상 문제가 되는 것은 아니다. 그러나 외부와의 인터페이스 역할을 하는 멤버함수가 중복되면 객체를 통해 호출하는 것이 모호해진다.

Derived클래스의 외부에서는 두 함수를 구분할 방법이 없는 것이 문제이다. 이 경우 범위연산자를 사용하여 기본클래스를 명시하면 원하는 함수를 호출할 수 있다.

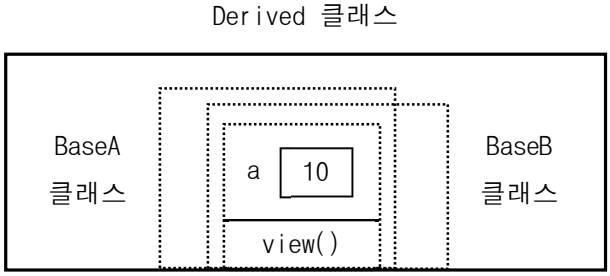
```
Derived d;
d.BaseA::view();  // BaseA의 view함수 호출 => Base(20) 출력
d.BaseB::view();  // BaseB의 view함수 호출 => Base(30) 출력
```

굳이 하나의 인터페이스를 원한다면 Derived클래스에서 view함수를 재정의하여 기본클래스의 함수들을 숨기는 것도 한가지 방법이 될 수 있다

```
class Derived : public BaseA, public BaseB {
public:
    void view(){  // view함수의 재정의
        BaseA::view();  // Base(20) 출력
        BaseB::view();  // Base(30) 출력
    }
};

Derived d;
d.view();  // 재정의된 view함수를 호출한다.
```

그러나 보다 근본적인 해결방법은 공통의 기본클래스를 하나만 포함하도록 상속하는 것이다. 중간단계의 상속에서 공통의 기본클래스를 **가상으로 상속받으면** 다중상속을 받는 파생클래스는 공통의 기본클래스가 중복되지 않도록 상속할 수 있다.
즉, BaseA와 BaseB클래스에서 Base클래스를 상속받을 때 virtual 예약어를 사용하여 Base클래스를 가상 기본클래스로 상속 받는다. 기본클래스를 가상으로 상속받는 것은 재상속되었을 때 중복을 차단하는 것이며 자신의 클래스는 기본클래스를 정상적으로 상속받는다.

```
class BaseA : virtual public Base {  // Base를 가상 기본클래스로 상속
public:
    BaseA() : Base(20) {}
};
class BaseB : virtual public Base {  // Base를 가상 기본클래스로 상속
public:
    BaseB() : Base(30) {}
};
class Derived : public BaseA, public BaseB {};
Derived d;
d.view();  // view를 재정의하지 않아도 직접 호출 가능. Base(10) 출력
```



가상 기본클래스를 사용하면 인터페이스의 중복문제는 깔끔하게 해결할 수 있지만 객체 생성과정에서 좀더 복잡한 문제를 일으킨다.
Derived의 객체는 그 기본클래스가 되는 BaseA와 BaseB의 객체를 먼저 생성해야 하는데 이들의 생성자는 다시 그 기본클래스가 되는 Base클래스의 생성자를 호출해야 한다는 것이다. 이 때

가상으로 상속받은 클래스들은 공통의 기본클래스 객체를 하나만 생성하고 서로 공유하여 기본클래스의 중복을 피한다.
그런데 문제는 BaseA와 BaseB 중에서 어떤 클래스의 생성자를 통해서 Base 객체를 생성하느냐에 따라서 초기값이 달라진다는 것이다. 따라서 컴파일러는 별도로 Base클래스의 디폴트생성자를 호출하여 Base클래스의 객체를 생성하게 된다.

그렇다면 만약 Derived클래스에서 Base클래스의 객체에 초기값을 전달해야 한다면 어떻게 해야 할까? 일반적인 상속에서는 파생클래스의 생성자에서 직계 기본클래스의 생성자를 명시적으로 호출하면 가능하다. 그러나 가상 기본클래스를 사용한 다중상속에서는 공통의 기본클래스의 객체가 항상 디폴트생성자에 의해 생성되므로 Derived클래스의 생성자는 BaseA나 BaseB의 생성자를 통해서 Base클래스의 객체에 초기값을 전달할 수 없다는 것이다.
이 경우 예외적으로 Derived클래스에서 직계 클래스가 아닌 Base클래스의 생성자를 호출하는 것을 허용한다.
중간 클래스에도 데이터멤버를 추가하고 각 생성자에 데이터가 전달되는 과정을 살펴보자.

[예제4-10] 가상 기본클래스를 사용하는 다중상속 프로그램

```
1:  #include <iostream>
2:  using namespace std;
3:  class Base{
4:  private:
5:      int a;
6:  public:
7:      Base(int n=10) { a=n; }
8:      void view() { cout << "Base(" << a << ')' << endl; }
9:  };
10:
11: class BaseA : virtual public Base {
12: private:
13:     int ba;
14: public:
15:     BaseA(int n, int m) : Base(n), ba(m) {}   // Base부분 초기화
16:     void view() {   // view함수의 재정의
17:         Base::view();   // Base부분 출력
18:         cout << "BaseA(" << ba << ')' << endl;   // 추가부분 출력
19:     }
20: };
21:
22: class BaseB : virtual public Base {
23: private:
24:     int bb;
25: public:
26:     BaseB(int n, int m) : Base(n), bb(m) {}   // Base부분 초기화
27:     void view() {   // view함수의 재정의
28:         Base::view();   // Base부분 초기화
29:         cout << "BaseB(" << bb << ')' << endl;   // 추가부분 출력
30:     }
31: };
32:
33: class Derived : public BaseA, public BaseB {
34: private:
35:     int dd;
36: public:
37:     Derived(int n, int ma, int mb, int l)
38:         : Base(n), BaseA(n, ma), BaseB(n, mb), dd(l) {}
```

```
39:         void view() {
40:             BaseA::view();   // BaseA 부분 출력
41:             BaseB::view();   // BaseB 부분 출력
42:             cout << "Derived(" << dd << ')' << endl;
43:         }
44: };
45:
46: int main()
47: {
48:         BaseA a(10, 20);   // Base(10), BaseA::ba=20
49:         BaseB b(30, 40);   // Base(30), BaseB::bb=40
50:         cout << "BaseA의 객체 출력..." << endl;
51:         a.view();
52:         cout << "BaseB의 객체 출력..." << endl;
53:         b.view();
54:
55:         Derived d(10, 20, 30, 40);
56:         // Base(10), BaseA(10, 20), BaseB(10, 30), Derived::dd=40
57:         cout << "Derived의 객체 출력..." << endl;
58:         d.view();
59:
60:         return 0;
61: }
```

실행결과

```
BaseA의 객체 출력...
Base(10)
BaseA(20)
BaseB의 객체 출력...
Base(30)
BaseB(40)
Derived의 객체 출력...
Base(10)
BaseA(20)
Base(10)
BaseB(30)
Derived(40)
```

BaseA와 BaseB클래스는 새로운 데이터멤버를 추가했으므로 각각 view함수를 재정의할 필요가 있다. 이 때 이들을 다중 상속하는 Derived클래스는 다시 두 개의 view함수를 갖게 되므로 역시 view함수를 재정의해야 한다. 그러나 재정의된 view함수에서 객체의 모든 데이터를 출력하기 위해 BaseA와 BaseB의 view함수를 호출하는 것은 공통의 기본클래스를 두 번 출력하는 문제가 생긴다. 따라서 다중 상속을 고려한다면 공통 부분과 추가된 부분을 분리하여 파생클래스에서 개별적으로 출력할 수 있도록 작성하는 것이 좋다.

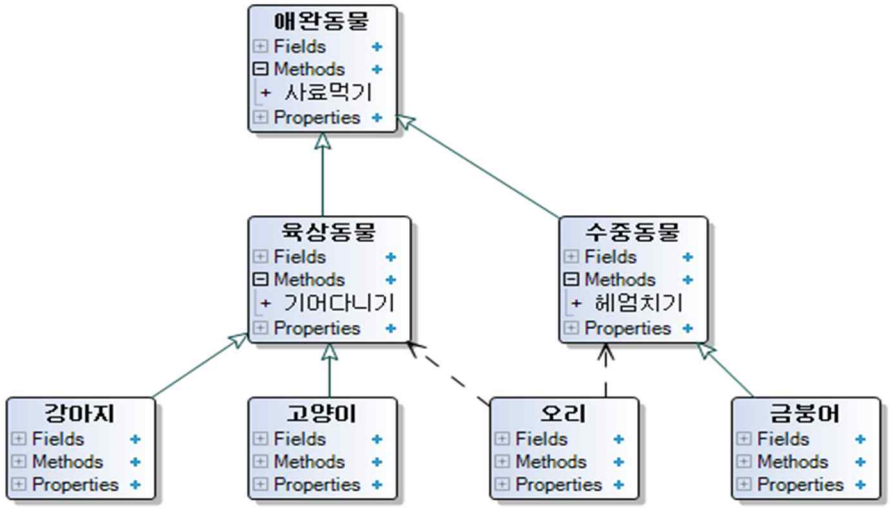
```
class BaseA : virtual public Base
{
private:
    int ba;
```

```
protected:  // 파생 클래스에서 사용할 목적이므로 protected 접근권한
    void viewA()  // 추가된 부분만 출력
    { cout << "BaseA(" << ba << ')' << endl; }
public:
    BaseA(int n, int m) : Base(n), ba(m) {}
    void view()
    {
        Base::view();
        viewA();
    }
};
class BaseB : virtual public Base {
private:
    int bb;
protected:
    void viewB()  // 추가된 부분만 출력
    { cout << "BaseB(" << bb << ')' << endl; }
public:
    BaseB(int n, int m) : Base(n), bb(m) {}
    void view()
    {
        Base::view();
        viewB();
    }
};
class Derived : public BaseA, public BaseB {
private:
    int dd;
public:
    Derived(int n, int ma, int mb, int l)
        : Base(n), BaseA(n, ma), BaseB(n, mb), dd(l) {}
    void view()
    {
        Base::view();
        BaseA::viewA();
        BaseB::viewB();
        cout << "Derived(" << dd << ')' << endl;
    }
};
```

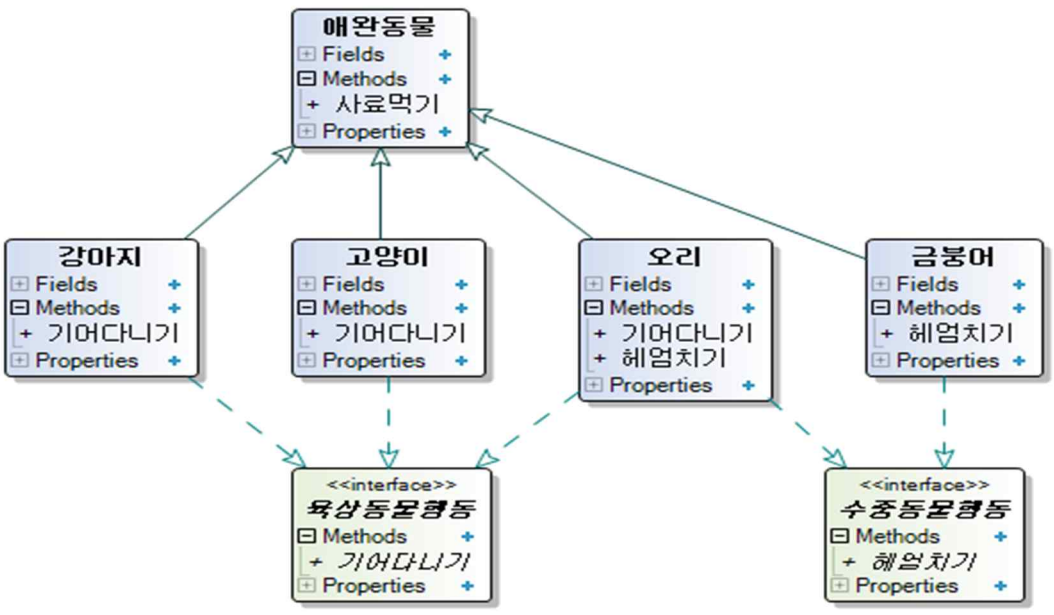
7. 인터페이스 개념

다중상속은 지금까지 살펴 본 바와 같이 두 개의 기본 클래스에 같은 멤버가 있는 경우 다중상속을 받고 사용하고자 할 때 어느 기본 클래스의 멤버를 호출하는가 모호성 문제 발생하는 등 사용에 어려움이 있다. 이런 다중상속의 문제점을 인터페이스 상속을 통해 해결한다.

(예) 오리는 육상동물 행동인 "기어다니기"도 하고, 수중동물인 금붕어와 같이 "헤엄치기"도 한다. 양쪽의 형질을 모두 이어 받으려면 다중상속이 필요하다. 다중상속은 대부분의 언어가 지원하지도 않고, 지원한다고 해도 무분별하게 사용하면 앞서 다중상속의 문제점과 같이 추후 감당하기 어려운 문제에 봉착하게 된다.
그림의 경우에는 상속이 두 단계에 거쳐서 일어나면서 모호성 발생 한다



인터페이스를 이용하므로 상속의 단계도 줄어들었고, 다중상속 문제도 해소할 수 있다.
"기어다니기"와 "헤엄치기"를 유전적 형질(is-a)로 보지 않고, 후천적 형질(can-do)로 보고 처리한 것 이다. (후천적 형질을 can – do 관계라고 한다)
다중 상속 인터페이스 구현은 can-do 관계를 통해 해결할 수 있다.



▌인터페이스(Interface)의 정의

C++는 순수 추상 클래스를 이용해 인터페이스 정의
- 클래스를 정의할 모든 멤버는 public 이므로 class 대신 struct 사용.
- 관례적으로 클래스 이름 앞에 접두사로 인터페이스 약자인 I 문자 포함.

```
struct IStack
{
        virtual ~IStack () { };
        virtual void push (void * item, int size) = 0;
        virtual void* pop() = 0;
};
```

인터페이스를 사용하는 이유
- 클래스 제공자와 사용자 사이의 강력한 규칙 제공
- 뛰어난 확장성

[ILandAnimal.h]

```
#pragma once
struct ILandAnimal
{
        virtual ~ILandAnimal() {};
        virtual void creep()=0;    // 상속받는 파생클래스에서 반드시 overriding을 해야 함
};
```

[IAquaticAnimal.h]

```
#pragma once
struct IAquaticAnimal {
        virtual ~IAquaticAnimal() {}
        virtual void swim() = 0;   // 상속받는 파생클래스에서 반드시 overriding을 해야 함
};
```

[Pet.h]

```
#pragma once
#include<string>
using namespace std;
class Pet   // 애완동물 클래스
{
public :
        typedef unsigned int age_t;
        enum PetKind { MAMMAL, FISH, BIRDS, UNKNOWN }; // 포유류, 어류, 조류, 모르는 종
private :
        string name;    // 애완동물명
```

```
        PetKind kind;   // 종류구분
        age_t age;      // 연령
public :
        Pet(string name = "noname", PetKind kind = UNKNOWN, age_t age=0) : name(name), kind(kind), age(age)
        {}
        virtual ~Pet() {}
        virtual void eat();    // "먹이를 먹습니다" 메세지 출력하기
        void view();  // 애완동물 정보를 한줄로 출력하기
};
```

[실행결과]

```
** Cat객체 테스트 **
고양이 먹이를 먹습니다
땅 위에서 사뿐사뿐 걷습니다
야옹이 (포유류) 2살

** Dog객체 테스트 **
개 먹이를 먹습니다
땅 위에서 네발로 걷습니다
멍멍이 (포유류) 5살

** Duck객체 테스트 **
오리 먹이를 먹습니다
땅 위에서 두발로 걷습니다
물 속에서 두발로 헤엄칩니다
도널드 (조류) 1살

** GoldFish객체 테스트 **
금붕어 먹이를 먹습니다
물 속에서 헤엄칩니다
니모 (어류) 2살
```

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)