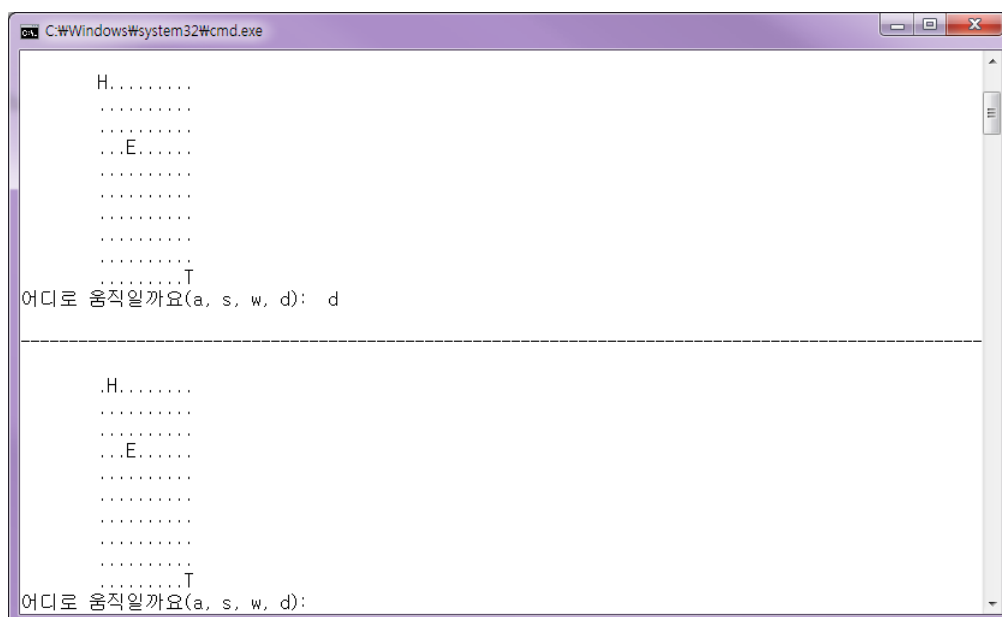


## 제12장 다형성과 가상함수

1. 다형성의 개념을 이해한다.
2. 상향 형변환의 개념을 이해한다.
3. 가상 함수의 개념을 이해한다.
4. 다형성을 실제로 적용할 수 있다.

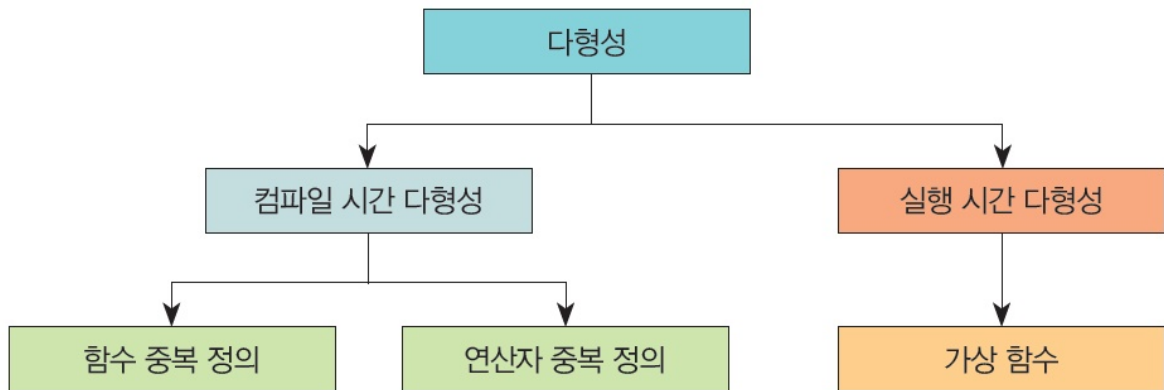
1

## 이번 장에서 만들어 볼 프로그램



2

## 12.2 다형성이란?



3

## 실행시간 다형성

- “실행 시간 다형성”이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 서로 다른 동작을 하는 것을 말한다.

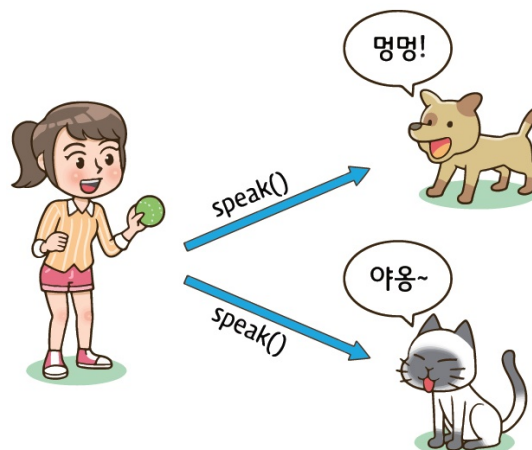
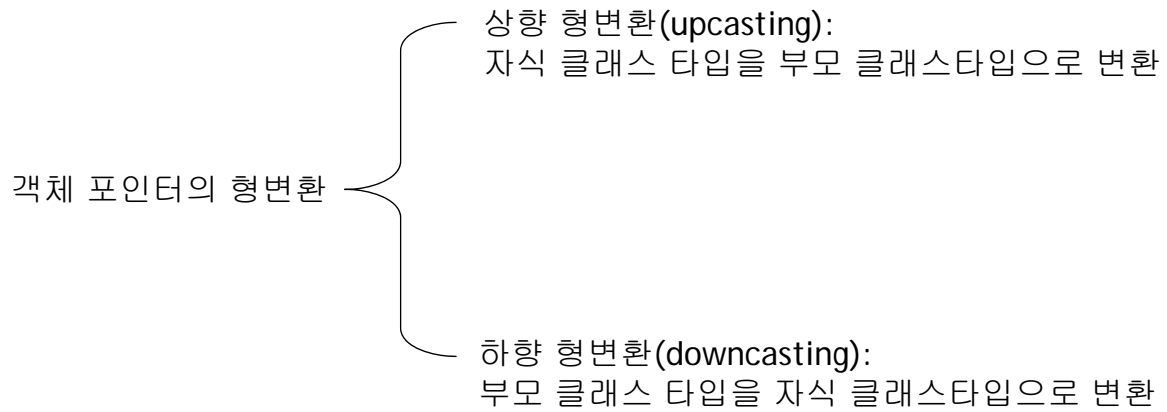


그림 12.1 다형성의 개념

4

# 객체 포인터의 형변환



5

## 상속과 객체 포인터

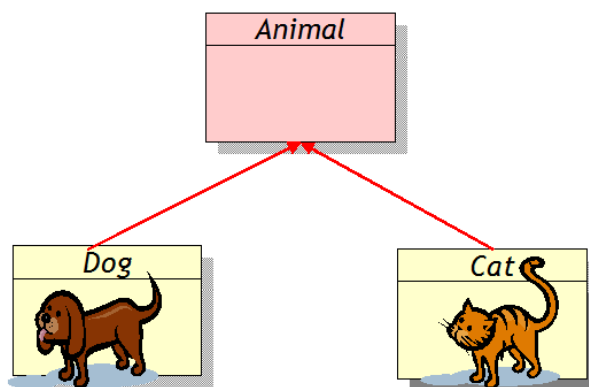


그림 9.2 상속 계층도

Animal 타입  
포인터로 Dog  
객체를 참조하니  
틀린 거 같지만  
올바른 문장!!

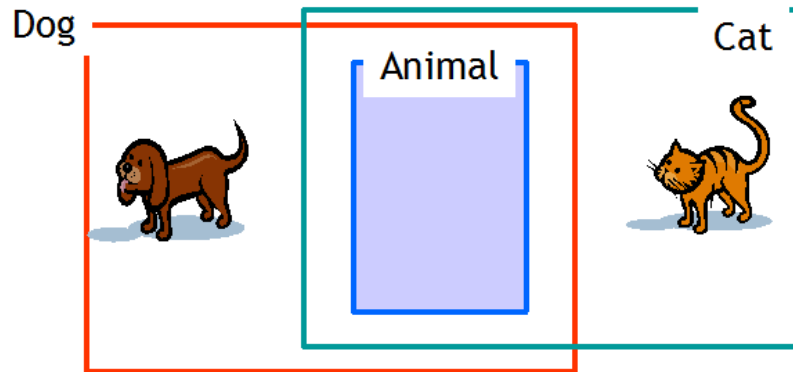
```
Animal *pa = new Dog(); // OK!
```



6

## 왜 그럴까?

- 자식 클래스 객체는 부모 클래스 객체를 포함하고 있기 때문이다.



7

## 상향 형변환

```
#include <iostream>
using namespace std;
class Animal
{
public:
    void speak() { cout << "Animal speak()" << endl; }
};
class Dog : public Animal
{
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};
class Cat : public Animal
{
public:
    void speak() { cout << "야옹" << endl; }
};
```

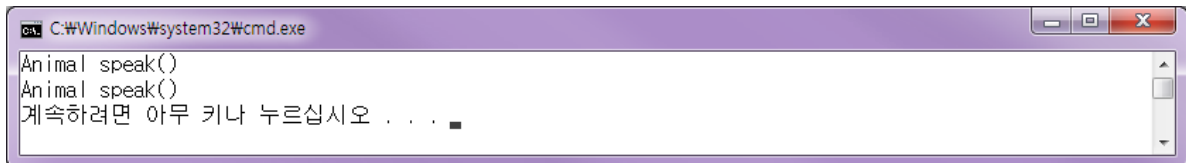
8

## 상향 형변환

```
int main()
{
    Animal *a1 = new Dog();
    a1->speak();

    Animal *a2 = new Cat();
    a2->speak();

    //a1->age = 10; // 오류!!
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Animal speak()
Animal speak()
계속하려면 아무 키나 누르십시오 . . .
```

9

## 기초 클래스의 포인터로 객체를 참조하면, (실습)

C++ 컴파일러는 포인터 연산의 가능성 여부를 판단할 때, **포인터의 자료형을 기준으로 판단하지, 실제 가리키는 객체의 자료형을 기준으로 판단하지 않는다.**

```
class Base
{
public:
    void BaseFunc() { cout<<"Base Function"<<endl; }
};

class Derived : public Base
{
public:
    void DerivedFunc() { cout<<"Derived Function"<<endl; }
};
```

```
int main(void)
{
    Base * bptr=new Derived(); // 컴파일 OK!
    bptr->DerivedFunc();       // 컴파일 Error!
    . . . .
}

int main(void)
{
    Base * bptr=new Derived(); // 컴파일 OK!
    Derived * dptr=bptr;      // 컴파일 Error!
    . . . .
}

int main(void)
{
    Derived * dptr=new Derived(); // 컴파일 OK!
    Base * bptr=dptr;             // 컴파일 OK!
    . . . .
}
```

10

## 앞서 한 이야기의 복습

“C++ 컴파일러는 포인터를 이용한 연산의 가능성 여부를 판단할 때, 포인터의 자료형을 기준으로 판단하지, 실제 가리키는 객체의 자료형을 기준으로 판단하지 않는다.” 따라서 포인터 형에 해당하는 클래스의 멤버에만 접근이 가능하다.

```
class First
{
public:
    void FirstFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void SecondFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void ThirdFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    tptr->FirstFunc();    (○)
    tptr->SecondFunc();   (○)
    tptr->ThirdFunc();    (○)
    sptr->FirstFunc();    (○)
    sptr->SecondFunc();   (○)
    sptr->ThirdFunc();    (×)
    fptr->FirstFunc();    (○)
    fptr->SecondFunc();   (×)
    fptr->ThirdFunc();    (×)
    . . . .
}
```

11

## 함수의 오버라이딩과 포인터 형 실행 FunctionOverride.cpp -> 실행 결과는?

```
class First
{
public:
    void MyFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void MyFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void MyFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}
```

FirstFunc  
SecondFunc  
ThirdFunc

함수를 호출할 때 사용된 포인터의 형에 따라서 호출되는 함수가 결정된다!  
포인터의 형에 정의된 함수가 호출된다.

실행: 각 함수 앞에 virtual 키워드 추가 후 실행

12

## 가상함수(Virtual Function) FunctionVirtualOverride.cpp

```
class First
{
public:
    virtual void MyFunc() { cout<<"FirstFunc"<<endl; }
};
```

```
class Second: public First
{
public:
    virtual void MyFunc() { cout<<"SecondFunc"<<endl; }
};
```

```
class Third: public Second
{
public:
    virtual void MyFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}
```

```
ThirdFunc
ThirdFunc
ThirdFunc
```

실행결과

포인터의 형에 상관 없이  
포인터가 가리키는 객체의  
마지막 오버라이딩 함수를  
호출한다.

13

## 실습 VirtualFunction.cpp-> 출력 결과는?

```
class CMyData
{public: virtual void PrintData() {cout << m_nData;}
    void TestFunc() { PrintData(); }
protected: int m_nData = 10;
}
class CMyDataEx : public CMyData
{public:
    virtual void PrintData() {cout <<m_nData*2;}
}
int main()
{
    CMyDataEx a;    a.PrintData();
    CMyData &b = a; b.PrintData();
    a.TestFunc();
    b.TestFunc();
}
```

14

## 가상함수

- **Animal** 포인터를 통하여 객체의 멤버 함수를 호출하더라도 객체의 종류에 따라서 서로 다른 **speak()**가 호출된다면 상당히 유용할 것이다.

15

## 예제

```
class Animal
{
public:
    virtual void speak() { cout << "Animal speak()" << endl; }
};
class Dog : public Animal
{
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};
class Cat : public Animal
{
public:
    void speak() { cout << "야옹" << endl; }
};
```

16

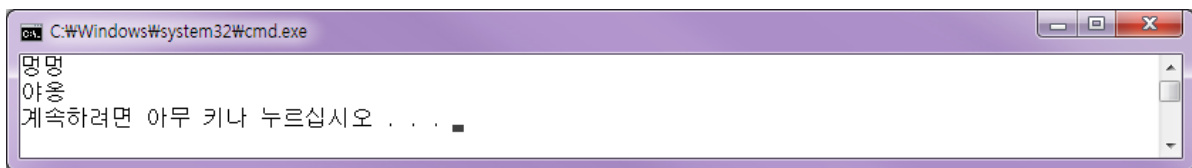


## 예제

```
int main()
{
    Animal *a1 = new Dog();
    a1->speak();

    Animal *a2 = new Cat();
    a2->speak();

    return 0;
}
```



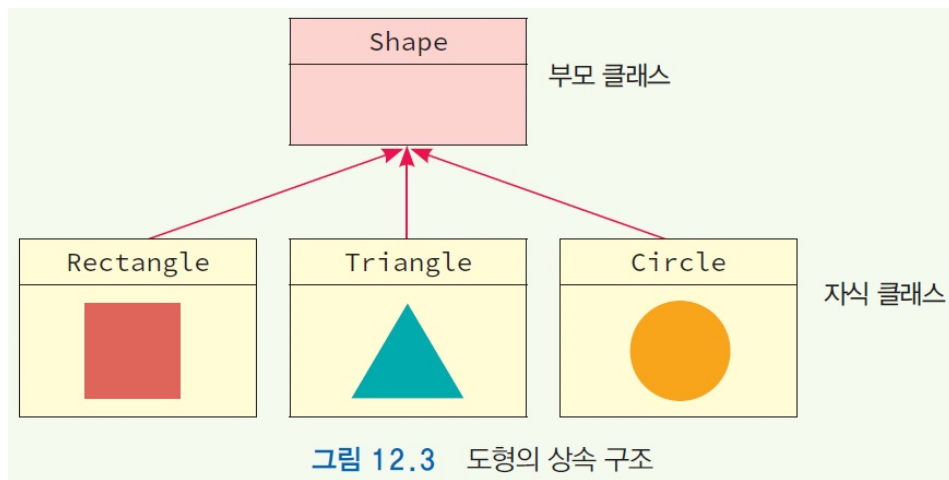
17

## 동적바인딩 vs 정적바인딩

바인딩의 종류	특징	속도	대상
정적 바인딩 (static binding)	컴파일 시간에 호출 함수가 결정된다.	빠르다	일반 함수
동적 바인딩 (dynamic binding)	실행 시간에 호출 함수가 결정된다.	늦다	가상 함수

18

## 도형 예제 #1



19

## 예제

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

class Shape {
protected:
    int x, y;

public:
    Shape(int x, int y) : x(x), y(y) { }
    virtual void draw() {
        cout << "Shape Draw" << endl;
    }
};
```

20

## 예제

```
class Rect: public Shape {
private:
    int width, height;

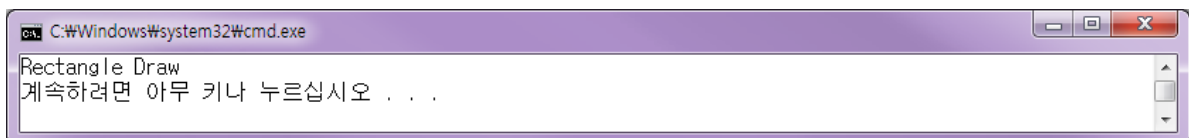
public:
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) {
    }
    void draw() {
        cout << "Rectangle Draw" << endl;
    }
};
```

21

## 예제

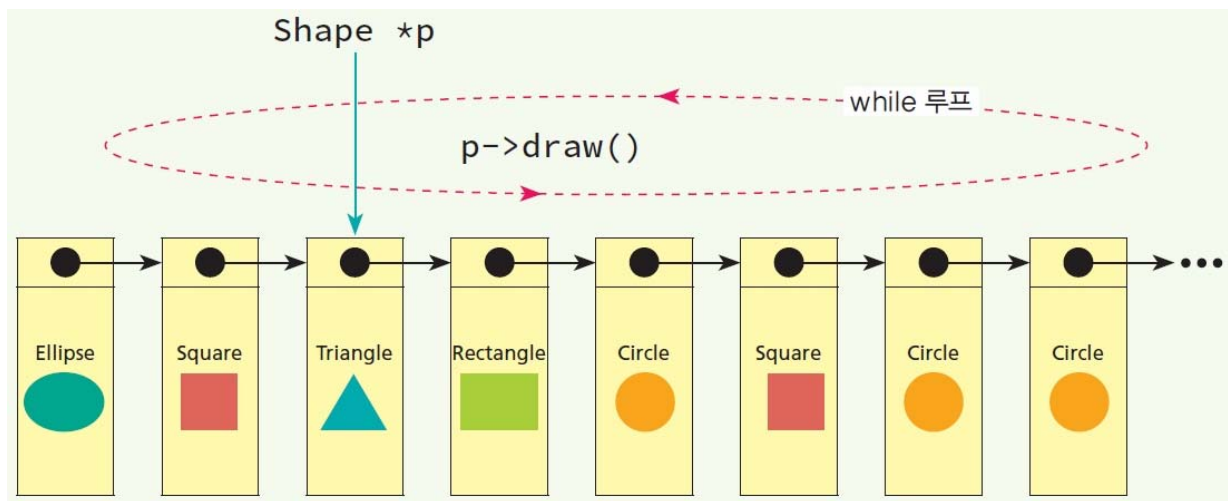
```
int main()
{
    Shape *ps = new Rect(0, 0, 100, 100);           // OK!
    ps->draw();

    delete ps;
    return 0;
}
```



22

## 도형 예제 #2



23

```
class Shape {
protected:
    int x, y;
public:
    Shape(int x, int y) : x(x), y(y) { }
    virtual void draw() {
        cout << "Shape Draw" << endl;
    }
};

class Rect : public Shape {
private:
    int width, height;
public:
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) { }
    void draw() {
        HDC hdc = GetWindowDC(GetForegroundWindow());
        Rectangle(hdc, x, y, x + width, y + height);
    }
};
```

24

```

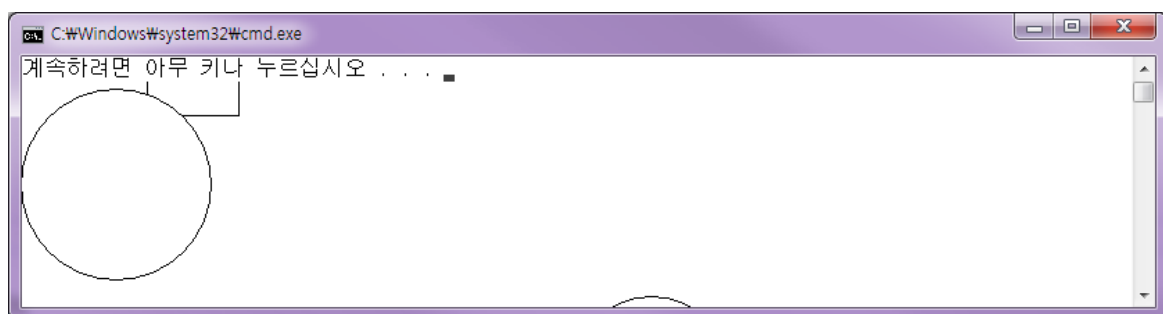
class Circle : public Shape {
private:
    int radius;
public:
    Circle(int x, int y, int r) : Shape(x, y), radius(r) { }
    void draw() {
        HDC hdc = GetWindowDC(GetForegroundWindow());
        Ellipse(hdc, x - radius, y - radius, x + radius, y + radius);
    }
};

int main()
{
    Shape *shapes[3];
    shapes[0] = new Rect(rand()%600, rand()%300, rand()%100, rand()%100);
    shapes[1] = new Circle(rand() % 600, rand() % 300, rand() % 100);
    shapes[2] = new Circle(rand() % 600, rand() % 300, rand() % 100);
    for (int i = 0; i < 3; i++) {
        shapes[i]->draw();
    }
}

```

25

## 예제



26

## 12.3 참조자와 가상함수

- 참조자도 포인터와 마찬가지로 모든 것이 동일하게 적용된다. 즉 부모 클래스의 참조자로 자식 클래스를 가리킬 수 있으며 가상 함수의 동작도 동일하다.

27

## final

- 가상 함수가 재정의되는 것을 방지
  - ▣ `virtual void PrintData() final;`
  - ▣ 함수 정의(선언 없을때)나 선언 모두에 사용 가능
  - ▣ 재정의시 컴파일 오류 발생
- 상속할 수 없는 클래스 지정에도 사용

```
class BaseClass final
{ };
class DerivedClass: public BaseClass // compiler error: BaseClass is
                                     // marked as non-inheritable
{ };
```

28

# 상속과 형변환

- `const_cast<>()`
  - ▣ `const` 포인터에서 `const` 제거
- `static_cast<>()`
  - ▣ 상속 관계의 포인터, 참조 상향 또는 하향 변환
  - ▣ 기초 타입의 형변환, 런타임 검사가 없음
- `dynamic_cast<>()`
  - ▣ 상속 관계의 포인터, 참조 안전한 상향 또는 하향(기초가 `polymorphic class`일 때만) 변환. 런타임 검사 수행.
- `reinterpret_cast<>()`
  - ▣ 조건없는 형변환

29

## static\_cast: A 타입에서 B 타입으로 StaticCasting.cpp

```
class Car
{
    // 예제 PowerfullCasting.cpp의 Car 클래스와 동일
};

class Truck : public Car
{
    // 예제 PowerfullCasting.cpp의 Truck 클래스와 동일
};

int main(void)
{
    Car * pcar1=new Truck(80, 200);
    Truck * ptruck1=static_cast<Truck*>(pcar1);    // 컴파일 OK!
    ptruck1->ShowTruckState();
    cout<<endl;

    Car * pcar2=new Car(120);
    Truck * ptruck2=static_cast<Truck*>(pcar2);    // 컴파일 OK! 그러나!
    ptruck2->ShowTruckState();
    return 0;
}
```

### `static_cast<T>(expr)`

포인터 또는 참조자인 `expr`를 무조건 `T`형으로 변환하여 준다.

단! 형 변환에 따른 책임은 프로그래머가 져야 한다.

`static_cast` 연산자는 `dynamic_cast` 연산자와 달리, 보다 많은 형 변환을 허용한다. 하지만 그에 따른 책임도 프로그래머가 져야 하기 때문에 신중히 선택해야 한다.

`dynamic_cast` 연산자를 사용할 수 있는 경우에는 `dynamic_cast` 연산자를 사용해서 안전성을 높여야 하며, 그 이외의 경우에는 정말 책임질 수 있는 상황에서만 제한적으로 `static_cast` 연산자를 사용해야 합니다

30

## static\_cast: 기본 자료형 간 변환

```
int main(void)
{
    int num1=20, num2=3;
    double result=20/3;
    cout<<result<<endl;
    ....
}
```

**C 스타일 형 변환**

```
double result=(double)20/3;
double result=double(20)/3;
```

**C++ 스타일 형 변환**

```
double result=static_cast<double>(20)/3;
```

static\_cast는 기본 자료형간 형 변환도 허용한다.

```
int main(void)
{
    const int num=20;
    int * ptr=(int*)&num;
    *ptr=30;    const 제거!
    cout<<*ptr<<endl;
    float * adr=(float*)ptr;
    cout<<*adr<<endl;
    ....
}
```

static\_cast 연산자는 '기본 자료형 간의 형 변환'과 '클래스의 상속관계에서의 형 변환'만 허용!

C언어의 형 변환 연산자는 원편에서와 같은 경우에도(모든 경우에) 형 변환을 허용. 따라서 제한적으로 허용하는 static\_cast 연산자가 훨씬 안정적이다.

상속과 관계 없는 포인터 형으로의 형 변환

31

## dynamic\_cast

### □ CastSample2.cpp

```
class CShape {virtual void Draw()...}
class CRectangle : public CShape {virtual void Draw()...}
class CCircle : public CShape {virtual void Draw()...}
CShape *pShape = nullptr;
pShape->Draw();    // 좋은 예
CRectangle *pRect = dynamic_cast<CRectangle*>(pShape);
CCircle *pCircle = dynamic_cast<CCircle*>(pShape);
```

32



## dynamic\_cast: 상속관계에서의 안전한 형 변환

DynamicCasting.cpp

```
class Car
{
    // 예제 PowerfullCasting.cpp의 Car 클래스와 동일
};

class Truck : public Car
{
    // 예제 PowerfullCasting.cpp의 Truck 클래스와 동일
};

int main(void)
{
    Car * pcar1=new Truck(80, 200);
    Truck * ptruck1=dynamic_cast<Truck*>(pcar1); // 컴파일 에러

    Car * pcar2=new Car(120);
    Truck * ptruck2=dynamic_cast<Truck*>(pcar2); // 컴파일 에러

    Truck * ptruck3=new Truck(70, 150);
    Car * pcar3=dynamic_cast<Car*>(ptruck3); // 컴파일 OK!
    return 0;
}
```

의도한 바 일수 있다. 그리고 이러한 down casting 경우에는 static\_cast 형 변환 연산자를 사용해야 한다.

dynamic\_cast<T>(expr)

포인터 또는 참조자인 expr을 T 형으로 변환하되 안전한 형 변환만 허용을 한다.

여기서 말하는 안전한 형 변환이란, 유도 클래스의 포인터 및 참조자를 기초 클래스의 포인터 및 참조자로 형 변환하는 것(up casting)을 의미한다.

33

## const\_cast: const의 성향을 제거하라!

ConstCasting.cpp

```
void ShowString(char* str)
{
    cout<<str<<endl;
}

void ShowAddResult(int& n1, int& n2)
{
    cout<<n1+n2<<endl;
}

int main(void)
{
    const char * name="Lee Sung Ju";
    ShowString(const_cast<char*>(name));

    const int& num1=100;
    const int& num2=200;
    ShowAddResult(const_cast<int&>(num1), const_cast<int&>(num2));
    return 0;
}
```

const\_cast<T>(expr)

expr에서 const의 성향을 제거한 T형 데이터로 형 변환하라!

포인터와 참조자의 const 성향을 제거하는 형 변환을 목적으로 함. 일반적 const 제거 아님.

함수의 인자전달 시 const 선언으로 인한 type 불일치가 발생해서 인자의 전달이 불가능한 경우에 유용함.

34

# reinterpret\_cast

- `reinterpret_cast < type-id > ( expression )`
  - ▣ Allows any pointer to be converted into any other pointer type.
  - ▣ Also allows any integral type to be converted into any pointer type and vice versa.

35

## reinterpret\_cast: 상관없는 자료형으로의 형 변환

실행 ReinterpretCasting.cpp

```
class SimpleCar { . . . . };  
class BestFriend { . . . . };
```

서로 아무런 관련이 없는  
두 클래스

`reinterpret_cast<T>(expr)`

`expr`을 `T` 형으로 형 변환하는데 `expr`의  
자료형과 `T`는 아무런 상관관계를 갖지 않는다.

```
int main(void)  
{  
    SimpleCar * car=new Car;  
    BestFriend * fren=reinterpret_cast<BestFriend*>(car);  
    . . . .  
}
```

형 변환의 결과는 예측하지  
못한다.

```
int main(void)  
{  
    int num=0x010203;  
    char * ptr=reinterpret_cast<char*>(&num);  
    for(int i=0; i<sizeof(num); i++)  
        cout<< static_cast<int>(*(ptr+i)) <<endl;  
    return 0;  
}
```

바이트 별 정수값 출력하기

`reinterpret_cast` 형 변환  
연산자의 적절한 사용의 예

3  
2  
1  
0

실행결과

36

## dynamic\_cast 두 번째 이야기: Polymorphic 클래스

### PolymorphicDynamicCasting.cpp

#### 형 변환 연산의 기본규칙

- 상속관계에 놓여있는 두 클래스 사이에서, 유도 클래스의 포인터 및 참조형 데이터를 기초 클래스의 포인터 및 참조형 데이터로 형 변환할 경우에는 dynamic\_cast 연산자를 사용한다.
- 반대로, 상속관계에 놓여있는 두 클래스 사이에서, 기초 클래스의 포인터 및 참조형 데이터를 유도 클래스의 포인터 및 참조형 데이터로 형 변환할 경우에는 static\_cast 연산자를 사용한다.

```
class SoSimple // Polymorphic 클래스!
{
    // ShowSimpleInfo가 가상함수이므로
public:
    virtual void ShowSimpleInfo()
    {
        cout<<"SoSimple Base Class"<<endl;
    }
};

class SoComplex : public SoSimple
{
public:
    void ShowSimpleInfo() // 이것 역시 가상함수!
    {
        cout<<"SoComplex Derived Class"<<endl;
    }
};
```

아래의 예에서 보이듯이 기초 클래스가 Polymorphic 클래스라면 유도 클래스로의 포인터 및 참조형으로의 형 변환이 허용된다!  
Polymorphic class: 하나 이상의 가상함수를 지니는 클래스

```
int main(void)
{
    SoSimple * simPtr=new SoComplex;
    SoComplex * comPtr=dynamic_cast<SoComplex*>(simPtr);
    comPtr->ShowSimpleInfo();
    return 0;
}
```

37

## dynamic\_cast와 static\_cast의 차이

기초 클래스가 Polymorphic 클래스라면 유도 클래스의 포인터 및 참조형으로의 형 변환에는 dynamic\_cast 연산자와 static\_cast 연산자 모두 사용할 수 있다.

하지만 여전히 dynamic\_cast 연산자는 안전성을 보장한다. 반면 static\_cast 연산자는 안전성을 보장하지 않는다.

```
class SoSimple // Polymorphic 클래스!
{
    // ShowSimpleInfo가 가상함수이므로
public:
    virtual void ShowSimpleInfo()
    {
        cout<<"SoSimple Base Class"<<endl;
    }
};

class SoComplex : public SoSimple
{
public:
    void ShowSimpleInfo() // 이것 역시 가상함수!
    {
        cout<<"SoComplex Derived Class"<<endl;
    }
};
```

```
int main(void)
{
    SoSimple * simPtr=new SoComplex;
    SoComplex * comPtr=dynamic_cast<SoComplex*>(simPtr);
    . . . .
}
```

형 변환 OK!

```
int main(void)
{
    SoSimple * simPtr=new SoSimple;
    SoComplex * comPtr=dynamic_cast<SoComplex*>(simPtr);
    . . . .
}
```

형 변환 실패! NULL 반환

dynamic\_cast 연산자는 위의 형 변환을 허용하지 않는다.

반면 static\_cast 연산자는 허용을 한다. 물론 그 결과는 보장받지 못한다.

38

## bad\_cast 예외

실행 DynamicBadCastRef.cpp

```
class SoSimple
{
public:
    virtual void ShowSimpleInfo()
    {
        cout<<"SoSimple Base Class"<<endl;
    }
};

class SoComplex : public SoSimple
{
public:
    void ShowSimpleInfo()
    {
        cout<<"SoComplex Derived Class"<<endl;
    }
};

int main(void)
{
    SoSimple simObj;
    SoSimple& ref=simObj;
    try
    {
        SoComplex& comRef=dynamic_cast<SoComplex&>(ref);
        comRef.ShowSimpleInfo();
    }
    catch(bad_cast expt)
    {
        cout<<expt.what()<<endl;
    }
    return 0;
}
```

Bad dynamic\_cast!

실행결과

참조자 ref가 실제 참조하는 대상이 SoSimple 객체이기 때문에 SoComplex 참조형으로의 형 변환은 안전하지 못하다.  
그리고 참조자를 대상으로는 NULL을 반환할 수 없기 때문에 이러한 상황에서는 bad\_cast 예외가 발생한다.

39

## 12.4 가상 소멸자

- 다형성을 사용하는 과정에서 소멸자를 virtual로 해주지 않으면 문제가 발생한다.

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

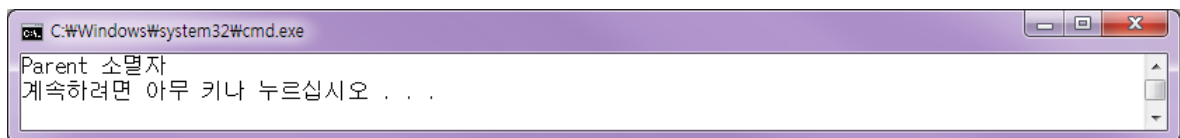
class Parent
{
public:
    ~Parent() { cout << "Parent 소멸자" << endl; }
};
```

40

## 가상 소멸자

```
class Child : public Parent
{
public:
    ~Child() { cout << "Child 소멸자" << endl; }
};

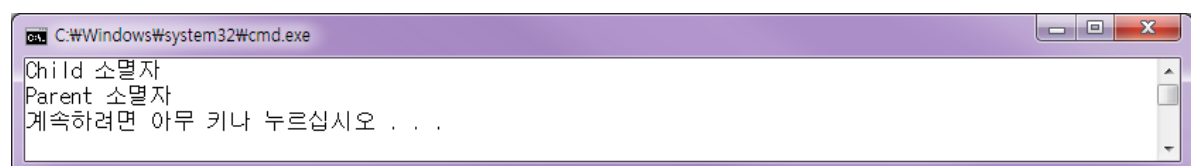
int main()
{
    Parent* p = new Child();    // 상향 형변환
    delete p;
}
```



41

## 해결책

```
class Parent
{
public:
    virtual ~Parent() { cout << "Parent 소멸자" << endl; }
};
```



42

## 가상 소멸자

---

- `CMyData *pData = new CMyDataEx;`
    - ▣ 상위 클래스로 하위 파생 클래스를 참조할 때의 상위 클래스 형식
    - ▣ 추상 자료형으로 동적 생성한 객체를 참조할 경우
      - 메모리 누수 오류 발생
      - 원인: 파생 형식의 소멸자가 호출되지 않음
      - `VirtualDestructor.cpp` 실행 -> 문제 해결하기
- ```
CMydata *pData = new CMyDataEx;
delete pData;
```

43

- 
- 소멸자를 가상 함수로 만들지 않으면 어떤 문제점이 발생하는가?
  - 어떤 경우에 반드시 부모 클래스의 소멸자에 `virtual`을 붙여야 하는가?

44

## 12.5 순수가상함수

```
class Shape {
protected:
    int x, y;
public:
    Shape(int x, int y) : x(x), y(y) {    }
    virtual void draw() = 0;
};
class Rect : public Shape {
private:
    int width, height;
public:
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) {
    }
    void draw() {
        cout << "Rectangle Draw" << endl;
    }
};
```

45

## 순수가상함수

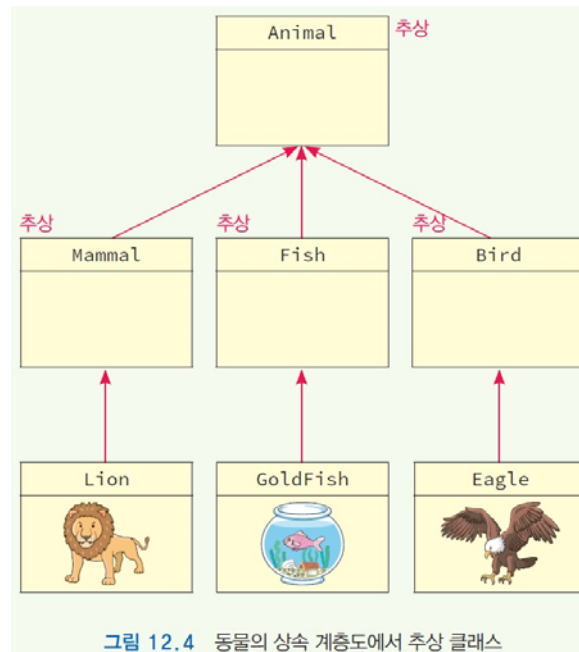
```
int main()
{
    Shape *ps = new Rect(0, 0, 100, 100);    // OK!
    ps->draw();                             // Rectangle의 draw()가 호출된다.
    delete ps;

    return 0;
}
```



46

## Lab: 동물 예제



47

## Solution

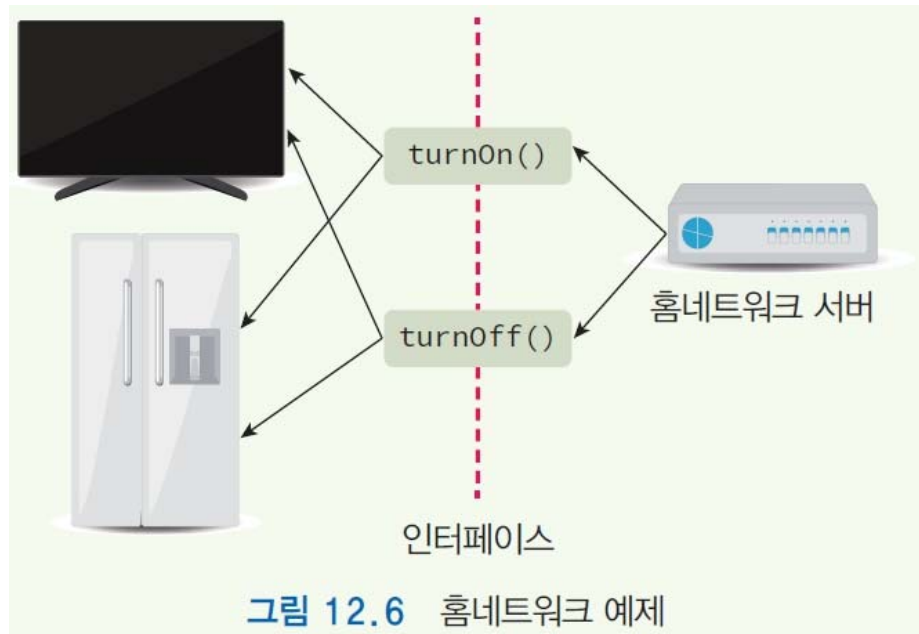
```
class Animal {
    virtual void move() = 0;
    virtual void eat() = 0;
    virtual void speak() = 0;
};

class Lion : public Animal {
    void move(){
        cout << "사자의 move() << endl;
    }
    void eat(){
        cout << "사자의 eat() << endl;
    }
    void speak(){
        cout << "사자의 speak() << endl;
    }
};
```

48



## Lab: 인터페이스



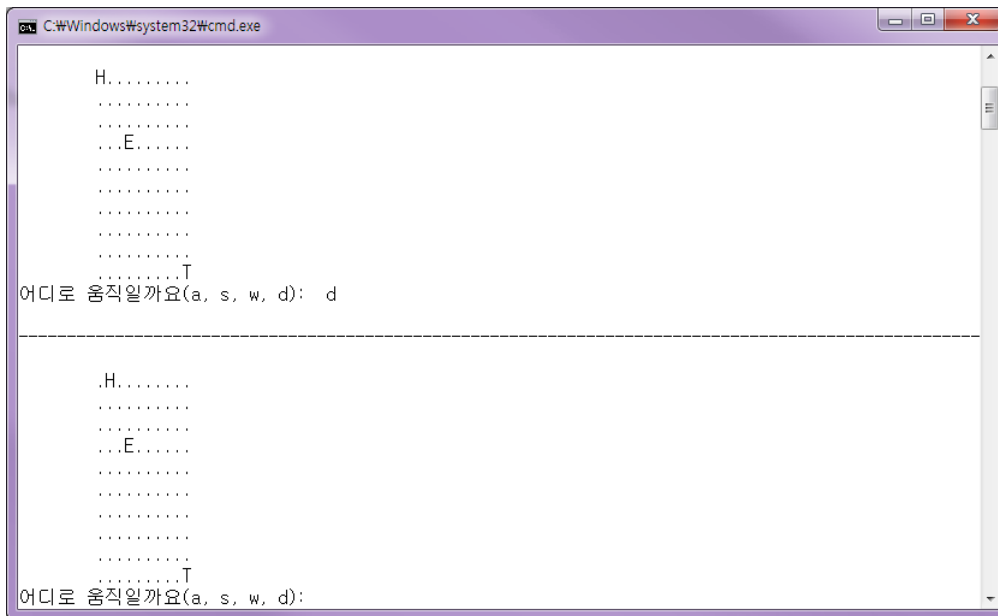
49

## Solution

```
class RemoteControl {
    // 순수 가상 함수 정의
    virtual void turnON() = 0; // 가전 제품을 켜다.
    virtual void turnOFF() = 0; // 가전 제품을 끄다.
}
class Television : public RemoteControl {
    void turnON()
    {
        // 실제로 TV의 전원을 켜기 위한 코드가 들어 간다.
        ...
    }
    void turnOFF()
    {
        // 실제로 TV의 전원을 끄기 위한 코드가 들어 간다.
        ...
    }
}
```

50

## Lab: 던전 게임



51

## Solution

```
class Sprite {
protected:
    int x, y; // 현재 위치
    char shape;
public:
    Sprite(int x, int y, char shape) : x{ x }, y{ y }, shape{ shape } {}
    virtual ~Sprite() {}
    virtual void move(char d) = 0;
    char getShape() { return shape; }
    int getX() { return x; }
    int getY() { return y; }
    bool checkCollision(Sprite *other) {
        if (x == other->getX() && y == other->getY())
            return true;
        else
            return false;
    }
};
```

52

## Solution

```
// 주인공 스프라이트를 나타낸다.
class Hero : public Sprite {
public:
    Hero(int x, int y) : Sprite(x, y, 'H') {}
    void draw() { cout << 'H'; }
    void move(char d) {
        if (d == 'a') { x -= 1; }
        else if (d == 'w') { y -= 1; }
        else if (d == 's') { y += 1; }
        else if (d == 'd') { x += 1; }
    }
};

// 보물을 나타내는 클래스이다.
class Treasure : public Sprite {
public:
    Treasure(int x, int y) : Sprite(x, y, 'T') {}
    void move(char d) {
    }
};
```

53

## Solution

```
class Enemy : public Sprite {
public:
    Enemy(int x, int y) : Sprite(x, y, 'E') {}
    void move(char d) {}
};

// 게임 보드를 표시한다.
class Board
{
    char *board;
    int width, height;
public:
    Board(int w, int h) : width{ w }, height{ h } {
        board = new char[width*height];
        clearBoard();
    }
    ~Board() {
        delete board;
    }
};
```

54

```

void setValue(int r, int c, char shape) {
    board[r*width + c] = shape;
}
void printBoard() {
    for (int i = 0; i < height; i++) {
        cout << "\t";
        for (int j = 0; j < width; j++)
            cout << board[i*width + j];
        cout << endl;
    }
}
void clearBoard() {
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            board[i*width + j] = '.';
}
};
void drawLine(char x) {
    cout << endl;
    for (int i = 0; i < 100; i++)
        cout << x;
    cout << endl;
}

```

55

## Solution

```

int main()
{
    // 벡터를 사용하여 게임에서 나타나는 모든 스프라이트들을 저장한다.
    // 다형성을 사용해야 하므로 포인터를 벡터에 저장한다.
    // 다형성은 포인터를 이용해야 사용할 수 있음을 잊지 말자.
    vector<Sprite *> list;
    int width, height;
    cout << "보드의 크기를 입력하시오[최대 10X10]: " << endl;
    cout << "가로: ";
    cin >> width;
    cout << "세로: ";
    cin >> height;
    Board board(height, width);
    list.push_back(new Hero(0, 0));
    list.push_back(new Treasure(height - 1, width - 1));
    list.push_back(new Enemy(3, 3));
}

```

56

## Solution

```
// 게임 루프이다.
while (true)
{
    // 보드를 다시 그린다.
    board.clearBoard();
    for (auto& e : list)
        board.setValue(e->getY(), e->getX(), e->getShape());
    board.printBoard();

    // 사용자의 입력을 받는다.
    char direction;
    cout << "어디로 움직일까요(a, s, w, d): ";
    cin >> direction;

    // 모든 스프라이트를 이동시킨다.
    for (auto& e : list)
        e->move(direction);
    drawLine('-');
}
```

57

## Solution

```
// 벡터 안의 모든 동적 할당을 해제한다.
for (auto& e : list)
    delete e;
list.clear();
return 0;
}
```

- 괴물이 랜덤하게 움직이게 하라.
- 괴물이 주인공을 잡으면 게임이 실패로 종료되게 수정하라.
- 주인공이 보물을 찾으면 게임이 성공으로 종료되게 수정하라.

58

## 상속 구조에서의 대입 연산자 호출 실행 InheritAssignOperation.cpp

```
class First
{
private:
    int num1, num2;
public:
    First(int n1=0, int n2=0) : num1(n1), num2(n2)
    { }
    void ShowData() { cout<<num1<<"", "<<num2<<endl; }

    First& operator=(const First& ref)
    {
        cout<<"First& operator=()"<<endl;
        num1=ref.num1;
        num2=ref.num2;
        return *this;
    }
};
```

```
class Second : public First
{
private:
    int num3, num4;
public:
    Second(int n1, int n2, int n3, int n4)
        : First(n1, n2), num3(n3), num4(n4)
    { }
    void ShowData()
    {
        First::ShowData();
        cout<<num3<<"", "<<num4<<endl;
    }
    /*
    Second& operator=(const Second& ref)
    {
        cout<<"Second& operator=()"<<endl;
        num3=ref.num3;
        num4=ref.num4;
        return *this;
    }
    */
};
```

부모 객체에  
자식 객체를  
직접 대입할  
수 있지만,  
반대 방향은  
안된다.

디폴트 대입 연산자는 기초 클래스의 대입연산자를 호출해준다. 그러나  
명시적으로 파생 대입 연산자를 정의하게 되면, 기초 클래스의 대입 연산자  
호출도 오른쪽의 예와 같이 별도로 명시해야 한다.

```
Second& operator=(const Second& ref)
{
    cout<<"Second& operator=()"<<endl;
    First::operator=(ref);
    num3=ref.num3;
    num4=ref.num4;
    return *this;
}
```

59

## 12.6 상속과 연산자 다중 정의

- InheritOperOver.cpp 실행 -> 에러 해결하기

```
class CMyData {
    CMyData operator+(const CMyData &rhs);
    CMyData& operator=(const CMyData &rhs);
    operator int();};
class CMyDataEx : public CMyData
    CMyData a(3), b(4);
    CMyDataEx c(3), d(4), e(0);
    e = c + d;    // error?
```

- 부모에서 정의된 대입연산자는 파생의 디폴트 대입연산자 안으로만 상속
  - ▣ 부모에서 정의된 대입연산자는 자식으로 바로 상속되지 않는다
  - ▣ 파생에 정의된 대입연산자에서는 직접 기초대입 불러야함
  - ▣ 파생 디폴트 대입을 부르려면 우측을 파생타입으로 변경
    - CMyDataEx operator+(const CMyDataEx &rhs) 정의
  - ▣ 단순 상속을 의도하면
    - 한계: 기초 클래스의 멤버만 접근할 수 있다
    - using CMyData::operator+;
    - using CMyData::operator=; //주체는 파생, 함수인수는 안바뀜
    - 파생클래스는 자신만의 연산자를 따로 가진 것처럼 작동

61

## 가상 상속

- 족보가 엉킨 상황
- VirtualInherit.cpp -> 문제?

```
class CMyObject
class CMyImage: public CMyObject
class CMyShape: public CMyObject
class CMyPicture: public CMyImage, public CMyShape
```

  - ▣ 가상 상속으로 기초가 한번만 포함되게 수정

```
class CMyImage: virtual public CMyObject
class CMyShape: virtual public CMyObject
```

    - 둘 다 virtual을 붙여주어야 함

62



## 객체지향 주소록

63

지금까지 배운 내용을 토대로 진정한 객체지향 프로그래밍에 도전!

- **C언어로 작성된 주소록 분석**

: C언어로 작성된 코드에서 UI부분과 자료구조 부분을 함수단위로 구분하고 코드를 분석한다.

- **필요한 클래스 설계**

: 프로그램을 이루는 요소를 규정하고 클래스로 구현한다. 중요한 점은 요소의 역할과 관계 규정이다.

64



## C 주소록 예제

리스트를 직접 다루는 함수 + 화면 및 사용자 인터페이스 함수

- 단일 연결 리스트 기반의 주소록 프로그램
  - ▣ 리스트: USERDATA 구조체
  - ▣ 연결 리스트를 직접 다루는 함수를 구별
  - ▣ 화면 출력, 사용자 인터페이스 등을 담당하는 함수 확인
- AddressBook.c 실행, 결과 확인
  - ▣ 주소 추가, 검색, 전체 출력, 제거, 종료 등 기본 기능을 구현하고 모두 실행하여 정상작동 여부를 확인한다.

65

## 첫 번째: 기초 수준 객체화

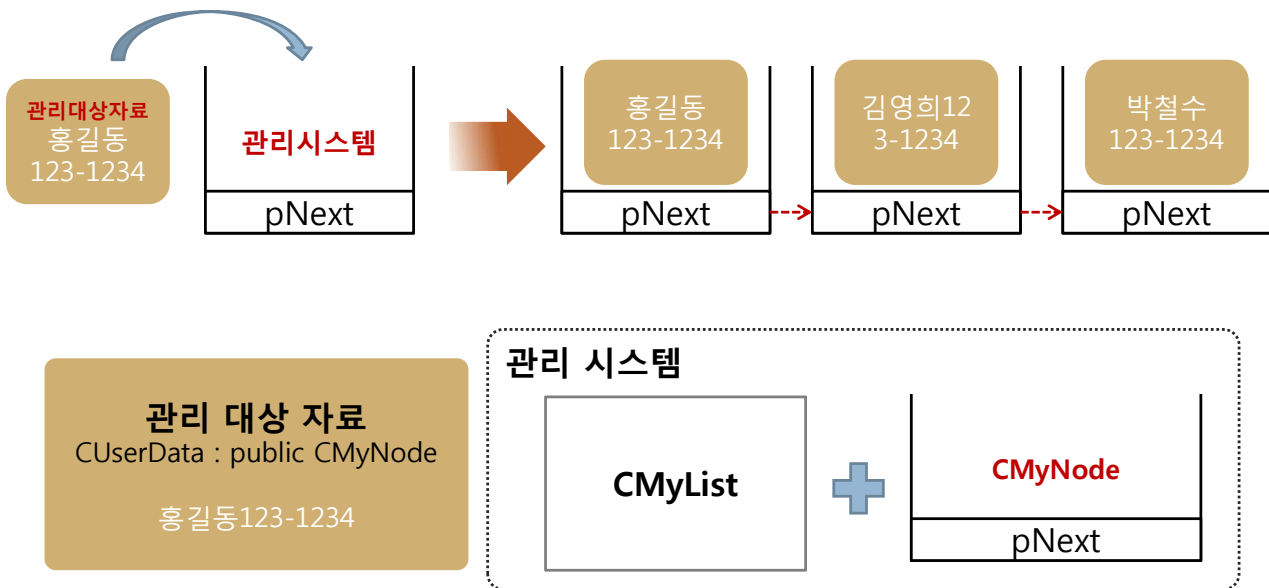
C언어로 작성된 예제를 다음과 같이 세 개의 클래스로 나눈다.

- C 함수를 세 개 클래스로 구성
  - ▣ CUserData: USERDATA 구조체와 동일한 의미를 갖는 클래스
  - ▣ CMyList: CUserData 클래스의 인스턴스를 단일 연결 리스트로 관리하는 클래스
  - ▣ CUserInterface: CMyList 클래스 참조를 멤버로 가지면서 사용자 인터페이스를 구현한 클래스
- AddressBook\_OOP 프로젝트 완성
  - ▣ CUserData: UserData.h, UserData.cpp
  - ▣ CMyList: MyList.h, MyList.cpp
  - ▣ CUserInterface: UserInterface.h, UserInterface.cpp
  - ▣ main(): AddressBook\_OOP.cpp

66

## 두 번째: 컨테이너 구현

CMyNode 클래스를 구현하고 자료구조와 대상 데이터를 분리한다.



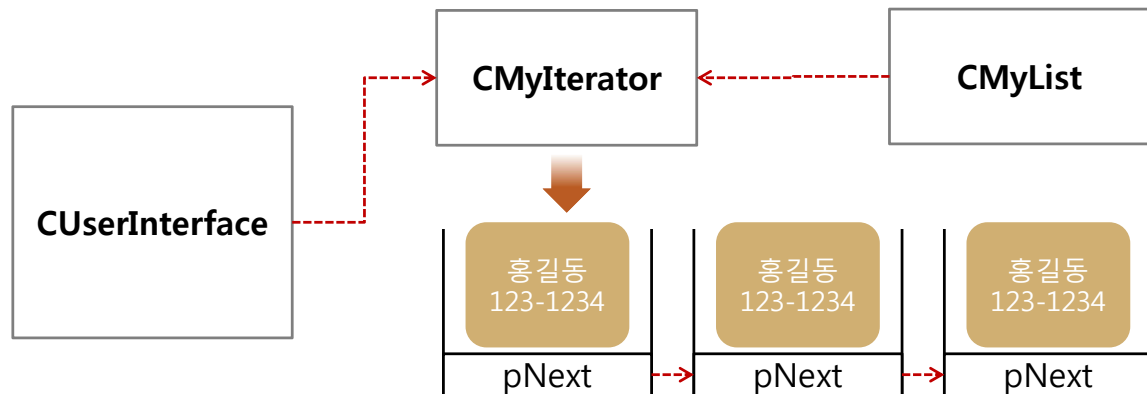
67

- '관리 대상'과 '관리 시스템'의 분리
  - 관리 대상: 자료 CUserData (CMyNode의 파생 클래스)
    - CMyNode의 파생 클래스라면 그 어떤 것이라도 CMyList에 의해 관리될 수 있다
    - CMyMovie 클래스(CMyNode의 파생)를 만들면 영화 관리 프로그램
  - 관리 시스템: CMyList + (CMyNode & pNext)
- 컨테이너 구현 : AddressBook\_OOP2 프로젝트 구현
  - CMyNode 클래스 추가와 관련 부분 후속 조치

68

## 세 번째: 반복자 구현

CMyList가 아닌 외부 클래스에서 연결 리스트 전체 데이터에 접근할 수 없도록 차단.



69

- 반복자 (Iterator)
  - CUserInterface -> CMyIterator <- CMyList
  - CMyIterator 클래스 추가: AddressBook\_OOP3 프로젝트 구현

70

## 과제

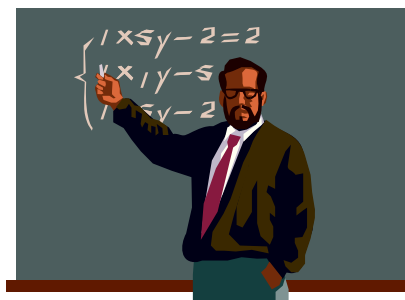
---

- 신규 관리 프로그램 작성 (애완견 관리, ...)
  - AddressBook\_OOP2 기반
  - 자료 멤버 2개 이상 추가
  - 관리 메뉴 4개 이상(삭제, 파일I/O 포함)
  - 자료는 파일 입출력으로 관리
- 제출 및 발표
  - ppt(프로그램 소개), 소스코드, 실행 캡처 화면
  - 시연

71

## Q & A

---



72